"Science isn't about why.
It's about why not!"
- Cave Johnson (Portal 2)

University of Alberta

Pinball: High-Speed Real-Time Tracking and Playing

by

Adam Metcalf

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Fall 2011
Edmonton, Alberta

Abstract

Pinball is fast-paced arcade-style game of which the origins date back hundreds of years. Game playing robots exist for billiards, foosball, and soccer and each have their own unique challenges. The speed that balls move in pinball machines requires that players have quick reactions.

We created a framework for Artificial Intelligence research using a Lord of the Rings pinball machine. Communication with the pinball machine is accomplished through an interface implemented in pinball games created with the University of South California's pinball controller. The framework is built around a real-time vision system for ball detection that uses a physics simulator to filter results. A network multi-player pinball protocol was created which opens new directions for creating new pinball games. To test the framework we recorded how long players can keep a single ball in play. This is the first attempt at building a high-performance pinball-playing robot.

Acknowledgements

Thank you Daniel Wong, Sven Koenig and the University of Southern California for making the pinball controller board available, as well as for your assistance with this project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Pinball is a classic game played on physical hardware. The basic premises of pinball is moving a ball around a slanted play field as seen in Figure 1.1. Different pinball machines have varying design, making each machine a unique game. Traditional pinball machines have a plunger and two flipper buttons. The flippers typically sit at the bottom of the play field and await a chance to interact with the ball. To start the game, a chrome ball is launched by the player pulling and releasing a plunger that sends the ball into play. There are many switches in the play field that are triggered by the ball. When switches are triggered they can assign points to the player, turn on lights, and change other switches' behavior. It can be played by one player at a time, but players can take turns to compare scores to make it a two-player serial game. The experts of the game are characterized by reflex speed, excellent timing, and planning. Most pinball machines are found in arcades or in the homes of pinball aficionados.

1.1 History of Pinball

The origins of pinball date back to the 17^{th} century. At the time many outdoor games were being transformed into tabletop games. Croquet brought about billiards, a similar game, but one which could be played inside. Bagatelle was also one of these evolutions. In France at the Château de Bagatelle, a game was played at a party honoring the King and Queen [5]. This game involved an ivory ball and obstacles in a play area. Players would use a small cue, similar to billiards, and shove a ball into the play area. The area had obstacles that would cause the ball to bounce when hit. When the ball came to rest it would reside in one of the scoring areas. The scoring areas were lowered sections of the table or rings of pins with the top of the ring open so the ball could roll into it. After the party, the game was named Bagatelle after the house at which it was popularized. The game became popular in France and then America when French soldiers brought their bagatelle tables over during the American revolutionary war.

Figure 1.1: The *Lord of the Rings* pinball machine playing field.

In 1871 Montegue Redgrave, a British inventor, received a patent in the United States for "Improvements in Bagatelle", which brought a spring propelled plunger and inclined play surface to the French tabletop game. These features allowed the table to be made smaller. Since then, the plunger and inclined surface have been used in the tables. Now that the game was on an incline a ball in the game would always try to reach the bottom of the area. In pinball the bottom play area funnels balls into the drain. Once the ball goes down the drain it is no longer in play.

The next big step came with the introduction of a solenoid[1] to the play area. In 1933, a game named *Contact* used a solenoid to propel the ball out of a bonus hole and another to ring a bell. The solenoids added feedback and excitement to the game, as well as made

[1] A solenoid is a coil of wire around an iron core that becomes a magnet when electrical current passes through the coil.

the game more dynamic. It was not until *Humpty Dumpty* in 1947 when the first player controlled solenoid flippers were added to the game. Now the players had more influence on the game and were actively engaged in playing rather than only watching their launched ball. The flippers made pinball a game of skill.

As players learned the tables, they also learned special tactics that could be employed to gain more control over the game. In particular, cradling or trapping was key to having control over the ball. To perform a cradle, the player would have to have the ball sit still in an engaged flipper. The player could then release the flipper and the ball would start to slowly roll down the flipper. Eventually a player could make specific shots by learning the time to wait before firing the flipper again. An even more advanced tactic is to pass the ball from one flipper to the other. This is highly beneficial as some shots are not possible to make with the left flipper but are with the right, and vice versa. Some skills involve bumping or nudging the machine in order for the ball to fall down the path the player wants. Bumping can sometimes save the ball or just earn the player extra points. However, some tables developed counter-measures for bumping to prevent aggressive use. If the table was bumped too much some pinball machines would penalize the player by locking out the flippers until the ball fell down the drain. Pinball experts have a toolbox of skills and they pair it with quick reactions to control the ball.

Pinball had a boom after World War II and was very popular but in the 1980s arcade centers started to replace pinball machines with video game arcade cabinets. To compete, manufacturers tried to incorporate the abilities of the computers into their pinball machines by adding displays and sounds. In this time they added many of the features we see in modern pinball machines such as ramps, blinking lights and multi-ball. However, this was not enough and the market for pinball machines greatly declined. Pinball manufactures often tried to tie-in their newest pinball machines to popular culture. They would license movies and television programs and design games related to them to draw a player in.

Unfortunately the pinball machine business does not see many new releases today. The old machines are collectibles among aficionados. However, pinball is not dead. As computers and video game consoles became approachable to consumers, they were able to bring the arcade game onto their televisions and monitors. The processors were capable of basic physics simulation that was required to play the game. Throughout video gaming history virtual pinball games have been released. *Video Pinball* on the Atari 2600 (1980) was the first video console pinball game. Soon after came the *Pinball Construction Set* (1983) which allowed players to create their own virtual pinball machines and play them. Many more have been created over the years. It is more affordable to buy a video game than to buy a machine.

Today, there are many ways to play pinball at home. *Visual PinMAME* [13] emulates

Figure 1.2: *Visual Pinball* running the *Lord of the Rings* table with *Visual PinMAME* running the *Lord of the Rings* game on a Windows 7 PC.

the pinball controller hardware and can be played from a single pre-rendered view point in *Visual Pinball* [11]. The software of commercial pinball machines is dumped into binary files for emulation. In Figure 1.2, the *Lord of the Rings* game is running inside *Visual Pinball* and the game logic and dot-matrix display is emulated by *Visual PinMAME* from the binary dumps. In the top left of the window is a rendering of the dot matrix display that appears exactly as it does on the actual machine. The game works as if you had the actual machine, down to the way the lights will flash, except it is all done in software. The physics simulation in *Visual Pinball* seems slower compared to what would be experienced in the real world.

The newer video pinball games take advantage of the latest 3D rendering hardware to create deeper immersion, as well as use the Internet to provide global score leader boards. Pinball is still popular and continues to live on in a simulated world.

1.2 Motivation

Since the amount of interaction the player has is limited to a few options, mainly pulling the plunger and firing flippers, pinball has been investigated as a testing ground for Artificial

Intelligence (AI) algorithms [15][28].

If a simulated (video) pinball game is used, the player has many advantages in this domain. Physical pinball machines have variance in places the players do not expect. Activating a flipper might actually take a few milliseconds longer than they expect. Solenoids might be warmer than usual and more powerful than when they are colder. These small changes can affect the outcome of the game. Getting a feel for the machine specifics, as well as reflexes, can be part of what makes pinball experts good at the games.

Not only do video pinball games lack the variance of a real machine, because the ball is virtual, its exact position and velocity are known. The movement of the ball in a pinball game is dynamic. A ball is eighty grams with a $1\frac{1}{16}$" diameter. The chrome balls get pushed around by large solenoids. In regular play, it is typical for a ball to change direction and actually gain speed after hitting a bumper. A video pinball AI agent would know where the ball(s) are at all times, but on a real pinball machine you cannot. The ball can even be completely obscured from the player by ramps and decorations.

Vision is an important real-world problem. Building a computer-controlled pinball player and being able to achieve high performance is a challenging task. Computer vision requires heavy image processing just to see objects, let alone track and observe their patterns.

With the assistance of a custom hardware controller board, a device that allows a developer to create their own pinball games to run on the physical pinball machine, we have total control of a real pinball machine. We will use a *Lord of the Rings* (LOTR) pinball machine, as seen in Figure 1.1, as a new test bed for AI research.

1.3 Problems & Solutions

Much effort is required to enable artificial intelligence (AI) research on a physical pinball machine. The first thing we need is a way for an agent to control the machine. We can accomplish this by using a hardware controller created by University of Southern California (USC) researchers [29]. The hardware controller interacts with a software library that gives a developer total control of the machine, including the state of the switches, firing solenoids, firing the flippers, and drawing on the dot-matrix display. This controller was originally designed to create new games. We extend the library to include a network protocol that emulates a human player hitting one of the switches.

The second requirement is being able to find the ball. We position a camera above the play field to accomplish this. Having the software simply detecting a ball is not enough; it has to be fast at doing so. The game has a ball that rapidly changes direction, can be obscured, and often can blend into the background. We must locate the ball(s) and try to keep track of and anticipate its movement. A system for doing this was built based around OpenCV [27], an open source vision library, which implements many of the best image

processing algorithms and provides a portable and stable framework to build on.

The third problem is to be able to hit the ball and aim the shots. Hitting the ball requires timing when the flippers should fire to get the ball heading in the direction you want. This is assisted by learning and simulation. The need for simulation comes from our next problem.

The forth consideration for accomplishing AI research with pinball machines is that they can break down. They are electrical and mechanical devices. The machine will eventually need repairs or a tune up. We must try to limit the use of the machine as much as possible to minimize wear and maintenance. Repair can be expensive as these machines use special parts that are not as mass produced as they once were. Machine upkeep being a concern means we cannot use the try-an-action-and-see-the-results approaches of reinforcement learning as the state space is large. This approach to learning requires taking many actions blindly and recording what actually happened until an acceptable strategy emerges. The ball has free range of motion in a two dimensional area so there is a lot of positions to be learned. The ball can move at great speeds and this inflates the number of positions possible. A learning table must either clamp the speed or increase the size of the table. For each physical position an agent would need to learn the action to do; this would stress the machine greatly. We can use a physics simulation that models the pinball machine to limit the use of the actual machine for simple learning problems. We use the Box2D [6] physics library, another open source project, to do offline shot learning and anticipate where the ball should be for the vision system.

Finally, we must present the state of the game to an AI agent to allow them to make decisions quickly. We must also give that agent all the information that is necessary to make good decisions. In the future this could involve learning and providing the rules to a specific game. For now, a simple interface was created to allow agents to play the pinball game that is separated from the vision system and hardware controller. Our interface provides the position, estimated velocity of the ball, and the ability to trigger the flippers and launch a ball.

1.4 Contributions

We have produced (1) a real-time computer-vision based tracking system for a pinball machine environment. This system is capable of locating a ball in a complex and potentially changing background. To showcase the capabilities of the system we have created a purely reflexive player. We also constructed (2) a multi-player protocol for pinball games. This protocol is abstract and allows for the creation of brand new pinball games. The addition of multi-player on multiple machines to pinball enables new kinds of AI challenges and competitions for pinball.

The vision system is based on open source libraries for computer vision (OpenCV) and physics (Box2D). Our system implements multiple ways of performing object detection by providing two ways for removing the background of a scene and two ways of collecting foreground points into objects. The vision system also includes a simulator. This simulator increases the speed and accuracy of ball tracking by limiting objects detected in the scene to only the objects that are physically possible. This simulator also allows offline learning that extends the life of the costly pinball machine components. The OpenCV Mixture of Gaussian background subtraction algorithm was parallelized with threads to improve performance. However, the images we are processing are not large enough to see significant speed-up. The reflexive player is simple but demonstrates that it is possible for a computerized player to play a game of pinball.

The multi-player protocol is based on another open source library (protobuf) and is generic and easy to use to design games. The networking code has been hidden behind a C++ class and makes multi-player pinball more accessible to game designers. Multi-player pinball can be used to pit man versus machine or man versus man in a new way.

The combination of these contributions provide a complete and functional system that can be used as a testbed for exploring AI solutions to creating an electronic pinball wizard.

Chapter 2

Related Work

The idea of a robot playing a game has been investigated in several scenarios. It poses an interesting challenge for vision, robotics, and artificial intelligence research. Here we report on the related work for pinball, foosball, billiards and soccer.

2.1 Video Game Pinball AI

Pinball is a popular game but few own an actual machine. The cost is prohibitive and has real-world complications. Some previous work has accessed the pinball domain by using a virtual video game world.

2.1.1 Pinball Algorithms

Michael Johnson, as part of his undergraduate thesis from MIT, classified the levels of skill in pinball and described useful algorithms for pinball [15]. He also constructed an AI agent that was designed to be used with a computer vision system. The agent calculated the trajectory of the ball and would use that incoming trajectory to determine when to hit the ball by finding the time until it contacts a flipper.

Johnson describes pinball skill using four levels: the spastic flipper, the able flipper, the accurate flipper, and the expert flipper. The spastic flipper is the most basic level. The player simply flips the flippers as much as possible when the ball comes into range. This strategy is somewhat effective, however you are often dependent on luck and thus do not get the high scores. The next level of skill is the able flipper. The player starts to pick and choose what flipper to fire and slowly becomes more accurate at this level. The following level is the accurate flipper. Now the player is able to pick and choose the shots to make and be able to control the ball. At this level, the player starts to learn to cradle or hold the ball in the flippers. This gives the player more time to think about the shot they need to make and they can make those shots a majority of the time. Finally, the expert or pinball wizard is able to hit the ball accurately from almost every position in the game. The expert avoids

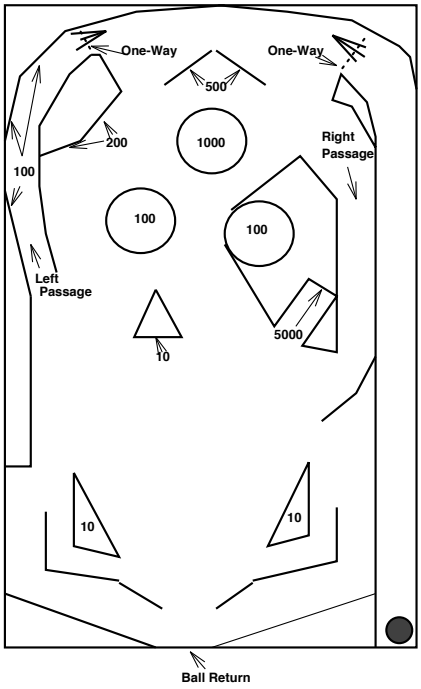

Figure 2.1: Simulator used by humans and AI agents in Johnson's experiments. Figure extracted from [15].

dangerous shots and knows many techniques for both achieving a high score in a game and keeping the ball from draining.

To test his algorithms Johnson built a simulator. Owning or building a pinball machine is costly. By having a virtual world he could not only test his algorithms but also have humans play the game to construct a benchmark for an average human player. The layout of the pinball table and the points for hitting objects can be seen in Figure 2.1. Johnson had many people play his pinball simulator at their leisure to gather statistics to compare against. The simulator was based on large time steps. The size of the step determines how computationally intensive the simulation is. At each instant the simulator would check if a collision has occurred between two objects. The simulator was limited to only having one moving object at a time. If a collision occurs the moving object will change direction by reflecting it about the normal of the surface it is colliding with and add some force if the surface was a bumper or flipper. This model did not measure the spin of objects. Johnson suggested that the benefit from modeling rotational velocity would be too low for the expected impact on performance. The simulator also clamps the ball speed at a maximum value to prevent it from moving too far during a single update. This is done to prevent an object from skipping through a surface without contacting it. Instead of reflecting off the surface, it teleports to the other side and continues with its same velocity.

This problem is solved in Box2D and will be discussed in Section 3.3.

The agents do not operate on the raw location data. Before the location data is sent to an agent or controller, the system does a trajectory projection. This projection estimates the distance the ball will travel to the flippers, which is used to determine the time until contact. This will allow the controller to plan shots if it knows how long it has until it needs to act. The controller is built up out of many mini-controllers which are in control of a partition of the flipper area. The partitions handle a range of angles in an ellipse that fits exactly around the flippers in both the fired and unfired configurations. Mini-controllers determine the time until impact and the speed of impact, and will return to the controller a time to hit and which flipper to fire.

Johnson attempted many different optimizations related to choosing the mini-controllers, however was not able to match the average human performance. He compared his controllers to the average human score results that he collected when people played his simulator. He noted that some mini-controllers would be penalized for things that are out of their control, such as a ball falling straight down the middle. As well, the plunger launching a ball would always end up in the same place and use the same mini-controller. The static launcher configuration caused that mini-controller to learn the first shot after being launched well but failed to perform in other situations.

Some assumptions of Johnson's simulator will not hold up in the real world on a pinball machine. Most important is the limit on the ball's velocity. Besides the simulator's limitations, the trajectory algorithms are still valid outside of the simulator and can be useful for pinball agents operating in the real world.

2.1.2 Reinforcement Learning

Nathaniel Winstead and Alan Christiansen [28] claim that a pinball machine with a camera vision system is a robot. They also claim that automated planning is required to achieve performance above the novice level. The authors describe two simple goals for a pinball robot: one is to keep the ball in play as long as possible (survival) and the other is to maximize the score for the game (score).

To test the automated planning the authors created a simulator. The simulator was based on simple collisions. The authors made a few more simplifying assumptions in their simulator. The incoming velocity of a ball is reflected about a normal vector of the object it is colliding with. As well, a small factor is added to the outgoing vector to account for loss of energy and random factors. They also assume that the flippers are infinitely fast, such that when a flipper button is pressed in the simulator, the flipper will contact the ball where it is at that instance of time and add a force to the ball from the contact. The layout of the simulated pinball machine is shown in Figure 2.2.

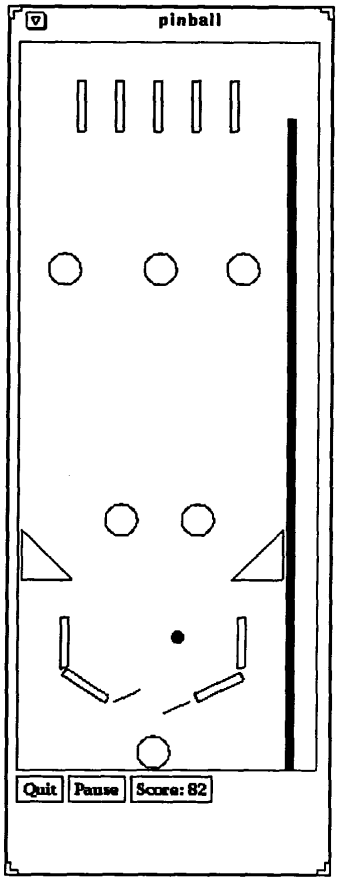

Figure 2.2: Pinball simulator layout used for learning by Winstead and Christiansen. Figure extracted from [28].

For planning, they divided up the area around the flippers into discrete space based on their position. Each cell was also broken up by the ball's velocity. If the ball is in the planning area, the planner has two actions to choose from: flip both flippers or do nothing. Once a flipper reaches the apex of the stroke it will return to its normal position. The planner did not have the ability to flip a single flipper at a time. Having two extra options for firing each flipper individually would slow overall learning time. It is acceptable to not have these extra actions when the flippers have infinite speed. The flippers can get out of the way as needed and act as if only one flipper was firing. Their planner does not allow for cradling or holding the ball in the flippers which is a much more complicated problem.

To seed the planner a basic Q-Learning [26] algorithm is used. They ran 10,000 balls through the agent, for both survival and score goals, to build the table of actions. The results showed that the difference between the survival and score maximizing goals was very little. Keeping the ball alive as long as possible enables you to get lucky and achieve more points. However, their result scores were not significantly different from a spastic flipper strategy, one of their baselines. The flailing strategy performs well based on their assumption of an

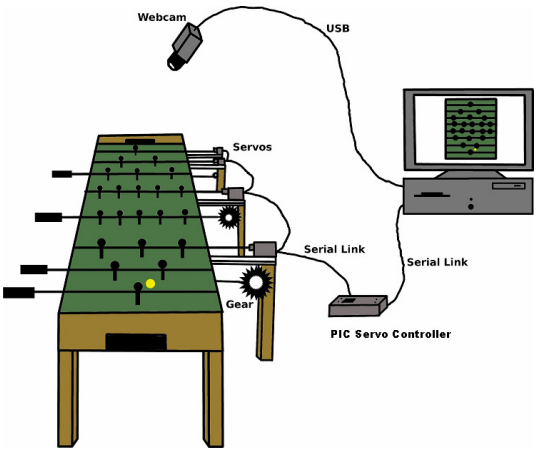

Figure 2.3: Overview of the automated foosball project. Figure extracted from [1].

infinite speed flipper and that there is not a physical robot that will break down if over used.

2.2 Foosball

Aeberhard et al., team of computing and electrical engineers at the Georgia Institute of Technology constructed an autonomous foosball table [1]. The goal of the project was to make foosball, which is a multi-player game, into a single player game under the constraint of a reasonable budget. They also wanted to make different levels of challenge so that it was approachable by newcomers and experts.

The foosball project fits in three main areas of research: vision, control, and AI. This is very similar to our pinball project. Figure 2.3 provides an overview of the system created to play foosball. First, a player must be able to understand and determine the state of the game. This will be accomplished with a webcam and software. Second, a player must be able to act in the game. For this a hardware controller will be used to drive motors and servos[1]. Lastly, a player must be able to determine and execute good decisions for what actions to take. An AI agent will run on a computer that will also process the images and communicate with the hardware controller to play the game.

Not much of the vision system algorithms are described in their report but it is implied that it relies heavily on connected component search based on color. At startup a player would select the color of the table's outline, the human controlled players, and the ball. For the outline it would find the rectangle of that color. Their vision system would then find a center of mass based on pixel colors for the foosball kickers and the ball. To help assist ball detection they use the ball's previous location to try and speed up the tracking of the

[1]An motor with an electronic input signal that is used to adjust the position of the armature or shaft.

following images. They specify a 20×20 pixel square around the ball's last position to find the ball. If the ball is not found, their vision system will scan the entire image looking for the ball color.

Much of this project was focused on the construction of the hardware controller board. The controller provides a protocol for controlling the game. A two-byte packet sent to the controller over a serial port would command the servos to move. The protocol also had special commands for tasks like centering all handles. The controller would decode these commands and provide power to the servos as requested.

A few issues arise from their implementation. The definition of objects based on color leaves their vision system vulnerable to poor lighting or changes in lighting. We believe it is required that they keep a lamp in a known location to be able to rely solely on color values.

To increase the performance of their vision system, they could have used predictions of velocity to shape the region of interest. This would save computation time when the ball is traveling at greater or less than 20 pixels per frame. We have come to a similar solution for pinball and this is discussed in Section 3.3.3.

In construction of the table Aeberhard et al. determined that many things did not meet their specifications. The webcam[2] underperformed and did not meet their requirements. The camera was advertised to provide 90 frames per second and 1280×1024 pixel images however this was not the case. They learned that the drivers for the camera would simply triple the same frame and quadrupedal each pixel to make it larger. This adds no new information and will only slow down image processing as the size of the images, as well as their frequency, will be increased. They were able to track the ball in 80% of the frames processed when they settled for 320×240 pixels per frame and 30 frames per second. We will discuss more on cameras and our camera selection in Section 3.2.

Aeberhard et al. also were not able to get the speed out of the servos that they required due to budget constraints. The kicking speed showed 1.5 feet per second and the movement speed was less than a foot per second. This made it hard to both get into the position needed and to kick the ball with any challenging speed. The ball would have to be moving very slowly in order for the foosball kickers to get into position first. Aeberhard et al. suggest more expensive servos are all that is required to fix these issues.

Finally, they suggest using an FPGA to do the processing instead of a computer. An FPGA is a customizable hardware device built up of logic gates that can be configured in a programming language. They claim that the latency between the visual event and the computer over USB is too much as it stands. An imaging sensor directly connected to a FPGA is a better solution, eliminating many sources of latency.

However, an FPGA adds more complexity to the design. We do not think this is nec-

[2] A consumer camera designed for computers for basic imaging problems.

essary. Cameras today are capable of achieving greater than 100 frames per second with IEEE 1394.[3] IEEE 1394 is isochronous, meaning that data comes regularly over equal time intervals. Due to the regularity of IEEE 1394 it can provide real-time data transfer. However, the higher quality cameras are more expensive than a consumer grade webcam but offer the performance needed for this project.

2.3 Billiards

Billiards shares some similarities to pinball. It is a game played on a surface and you interact with balls in a playing area. However, they are quite different in pace. Pinball is a fast paced game; in billiards you have time to make your shot as the game state is static. This slower pace allows an AI agent to take the time to think about the shots it needs to make to win. The time could allow an agent to use classical search algorithms to plan out its future states, provided an extremely accurate physics model could be made. The annual Computer Olympiad tournament uses a billiards simulator for their competitions. However, a robotic player is required to verify that it is possible to apply this research to the real world.

2.3.1 Billiard Playing Robot

Researchers around Ontario in Canada, led by Michael Greenspan, developed a pool playing robot [19]. This gantry robot has a ceiling mounted camera for positioning and second camera mounted on the robot for calibration. The second camera is used to improve the positional accuracy of the gantry robot.

There are many problems to solve before the robot can hit a ball. One of the problems for a robot player is constructing a mapping of a 2-dimensional camera image to a 3-dimensional coordinate that the robot can use. They construct an estimated 3×3 projection matrix using the ceiling mounted camera and the robot mounted camera. They use the balls in known positions on the table with a least-squares error estimate to build the projection matrix. This provides their system with a way to link a point from either camera to a point in the other.

To do this calibration they must first be able to locate the balls on the table. The images from the cameras are different. They do not share the same perspective of the table. The ceiling camera lens distorts the image in order to get the whole table in the frame. Because of this the side edges of the table look as though they are bent. A radial distortion filter is applied to correct for any configuration issues before processing the images from the ceiling camera. After they fix the perspective, the images are run through a simple threshold filter. This creates a binary image from the gray scale image. The pixels in the image that were

[3] IEEE 1394 (Firewire) is a alternate standard for computer-peripheral communication and is often used for cameras.

bright remain in the image as white and those that did not meet the threshold become black. The resulting image has some noise. The image is then processed with a connected component algorithm to remove groups of pixels that are too small or large to be a ball. Once the noise has been removed, only the true balls remain in the image as white circles. From these white circles they estimate the center of the ball with a least squared error.

The second camera is used to precisely locate the ball but it is a more difficult task. The camera position is not static and to determine the location of the ball the second camera will rotate around the ball. Using the location returned by the ceiling camera, the gantry robot will move over to the ball. After it is above the target, it will proceed to capture images while rotating around the ball. Using these images the center of the ball can be estimated but with much greater precision. As more images are taken from different positions around the ball the precision of the estimated center increases. When the center is located, the gantry robot will move by the difference in location between the previous estimation and the new one. This process will continue until the position meets the required accuracy. Their experiments show that this process requires at least 3 iterations to have 0.1mm accuracy.

The researchers then experimented with potting. Potting is the act of hitting a ball into a pocket on the table. In many billiards games this is called a scratch and is not a valid shot. The shot will have no collisions before entering the pocket. This is the most basic shot that one can perform when playing pool. They were able to achieve a 66% success rate in potting the ball. The failures occurred in certain regions due to some problems in the gantry positioning.

The main contribution by Greenspan et al. for billiards robotics was the improvement in positioning of a gantry robot by adding another camera. The single ceiling camera could not provide the level of accuracy needed to position or correct the robot. The second camera takes advantage of the static state of the game to use more time to achieve better performance. They achieved a 0.3mm average improvement in positioning and lowered the standard deviation by 0.5mm. In pinball we cannot use extra time to improve accuracy as the game can be changing with time. The game of pinball moves and requires quick reactions as well as precision.

2.3.2 PickPocket AI

Many billiards games are two-player turn-based games, unlike the potting game used for the gantry robot experiments. Having multiple players makes billiards adversarial, competitive, and allows AI agents to compete against one another. With the help of a robot like the one described in the previous section, it is possible for a robot to play a billiards game on a real table and eventually against a human opponent. Currently billiard AI competitions occur in a simulator. This is needed until the robots are able to have the precision needed and

also so anyone can participate in billiards AI research without investing in expensive gantry robot hardware.

PickPocket [24], authored by Michael Smith, is a two-time champion of computerized pool at the Computer Olympiad competition. The game that PickPocket plays is 8-ball, which is a very common billiards game. In this game the players try to sink their balls and finish with the solid black 8-ball. If a player manages to legally hit a ball into a pocket they get to continue with another turn at the table. If the player fails to down a ball on their turn or makes an illegal shot, such as downing the cue ball, then the opposing player gets a turn. This can create situations of one player acting many times in a row, downing many of their balls, and getting closer to winning the game.

Professional billiard players will try to leave the balls in positions where they are able to perform easier shots. These are shots where they have a high chance of continuing to play. This is similar to looking ahead in a classical search. While an easy shot might be good at the moment, that shot might leave the state of the game in a situation where the player has no way to continue to down additional balls and thus must give the opponent a chance to score and maybe win.

Not every shot is guaranteed to be made. Billiards is stochastic in nature; even professional players cannot have perfect accuracy with a cue stick. The professionals train to reduce the error and increase their accuracy. Some shots are easier than others as they are more forgiving of error. The error is based on many factors; including distance of the cue ball from a target ball, distance of the target ball to the pocket, angle of the target ball trajectory from the side rail, and angle difference in trajectory between the cue ball and the target ball. These four parameters, shown in Figure 2.4, describe any shot with a single collision between a cue ball and a target ball.

PickPocket simulated shots while changing these parameters with varying error to gather a probability of successfully pocketing the ball. This would determine which shots are more forgiving of error and therefor have a higher chance of being successful. The billiards physics engine used was *poolfiz* [18]. The simulator takes a game state (the position of the balls on the table) and a shot to perform and will return the game state after the shot has completed.

PickPocket uses *poolfiz* to create potential future table states that can be compared to others to determine which is better. Figuring out which states are better requires a good evaluation function. Pickpocket's evaluation function is based on the availability of shots as well as their quality. With many high quality follow up shots, PickPocket will have many options to choose from if it is able to continue. The evaluation function chosen is a weighted sum of probabilities of the potential shots. The weights decrease from the best shot to some limit. PickPocket only considers the three best shots in its weighted sum. This helps even out the values of a state that has many mediocre shots versus a state with one really easy

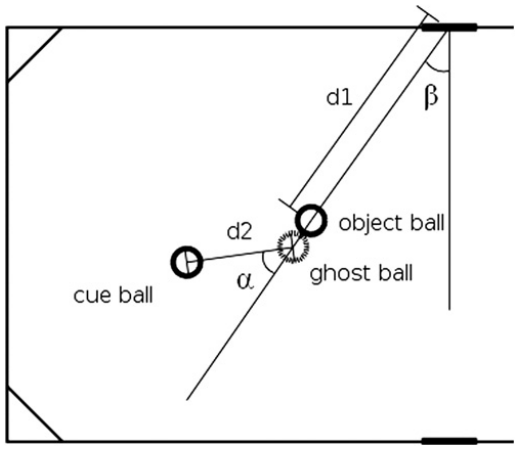

Figure 2.4: Diagram of the shot difficulty parameters of billiards. Figure extracted from Smith [24].

shot but very few other easy shots.

PickPocket searches through table states with a way to differentiate the states' values. Two search algorithms were used in PickPocket: Expectimax and Monte-Carlo search. Expectimax is classical search on a stochastic domain. The value used to determine if a move is good is the weighted sum of all its children. The weighting used is the probability of the shot being successful. Since billiards has possibly infinite positions PickPocket does not model the opponent's shots.

In billiards, it is possible to continue sinking balls and keep making shots until the end of the game is reached. We believe that the opponent will also try to sink balls and this will limit the number of shots considered by the opponent to a finite set, in a way similar to PickPocket. An agent could then use the opponent's shots in the evaluation function as a way to prevent an opponent from receiving good shots in case a shot fails.

Monte-Carlo search is a great tool for searching large state spaces. It does not exhaustively search through every state but instead runs multiple random simulations. In PickPocket, each available shot creates a new child node that represents an outcome. The shots are randomly perturbed by a noise model to create similar possible outcomes. The shots are then simulated multiple times and the score of a node is the average value of its children. The more simulations done from a particular state cause that states's value to converge to a more accurate representation of the shot. This tree is exponential and becomes quite large quickly, allowing for only simulation of shots at a maximum of 2 shots away.

When there are no good shots to take PickPocket looks at safety shots. These are shots that position the ball in a safe place. "Safe" means it is difficult for the opponent to sink a ball and this will hopefully end their turn quickly. PickPocket will simulate different shots with various angles and cue velocities to determine which shots are safe by evaluating the

resulting state from the opponent's perspective.

The results show that Monte-Carlo is a better search algorithm. Under low and mid range execution error, Monte-Carlo is better when compared to a greedy player that always does the shot with the highest success rate, and the Expectimax probabilistic search. Under high execution error probabilistic search and Monte-Carlo search gave comparable results but Monte-Carlo still won. It was discovered that the potential error on the shot has a massive impact on the benefit the depth of a search will have. Only under low error did searching deeper than one shot have a benefit. Search depth is an important thing to consider in stochastic games, like pinball. Executing a shot because a position many steps ahead is great is a poor decision in a domain where it is impossible to predict the exact outcome of a shot.

In 2008, PickPocket was defeated by CueCard [2] at the Computer Olympiad. CueCard was inspired by PickPocket and in one-on-one play CueCard was able to win 77.4% of the games. To win, CueCard's search was distributed across twenty computers with an optimized simulator that was five times faster than the previous version. The future of computer billiards is still very open. Future tournaments could have adjustable noise levels and play different games, such as 9-ball and snooker. A new framework has been constructed for the next tournament that is more usable and will lower the barrier of entry for new teams.

Billiards has a much different time constraint than pinball. Not only is pinball stochastic due to mechanical features such as solenoids, but it operates in real-time. A player will not have the ability to do forward planning from a static state unless the ball is cradled. We can learn from PickPocket and believe that from a cradled position it is not beneficial to plan future shots beyond that first hit. Not only will that shot have an error caused by mechanical elements but it will likely have more error introduced by other objects in the game. In billiards, players require the game state to settle before making their next action but in a pinball game it is not often that the ball will come to a rest, except when cradled.

2.4 RoboCup

RoboCup is an annual soccer tournament for robots. Their are many different events to showcase the latest innovations in artificial intelligence for robots, in both a simulation and physical league. Soccer is a multi-agent cooperation task. There are different class sizes of robots ranging from small to humanoid.

In the small league, the robots are tracked by an overhanging camera. This idea is described in Section 2.1 as a way to create a pinball robot AI. SSL-Vision [30] is a community effort to unify the vision framework for RoboCup soccer's small sized league. This framework aimed to reduce setup time between games, given that most teams converged to similar solutions for vision systems.

The framework is built upon QT, a platform-independent user interface library [7]. It employs a multi-threading model to make use of multiple cameras. In RoboCup, it is typical to want one camera over each half of the playing field to offer a clear picture of the entire area. The framework has a client that allows the user to view the game while it plays. In order to not inhibit the frame-rate of the image processing the creators constructed a circular buffer to store images. The critical image processing always operates on the latest image captures while the visualization thread can run in parallel and skip frames as needed to catch up to the latest images. This circular buffer parallelization prevents frame-rate loss caused by needing to wait for monitor refresh rates. It locks the images to render them before moving on. We implement this circular buffer model for our pinball vision system. (See in Section 3.2).

2.5 Conclusions

After searching the literature we discovered that there is not a lot of work in the area of creating a robotic pinball player. The technology exists today to make a real pinball wizard. Much of the research in artificial intelligence occurs in simulators but like RoboCup's soccer league, billiards gantry robots, and robotic foosball we must also show it is possible to accomplish this in the real world.

We expected to see an old arcade game-playing robot when attempting to find related work but found nothing. Older games like Atari's *Pong* or Namco's *Pac-Man* seem like excellent platforms for AI robots. These games have simple graphics and contrasting colors for different objects, making them easily extractable with a vision system.

There is still much to do before a computer controlled pinball machine can challenge a human pinball wizard's high scores. Pinball has a lot of opportunities for research. We list some ways to proceed with our research in Section 5.1.

Chapter 3

Architecture

We have designed a system made up of modules for a robot to play pinball. Each module has a specific task it must perform for the whole system to work. The modularity isolates responsibility and can allow algorithms to be swapped and tested easily. Figure 3.1 shows a overview of the modules and their relationships within the Pinball AI project.

The hardware of the pinball machine is modified by attaching a custom made controller. The controller, (a) in Figure 3.1, is used to enforce the game definition. It is responsible for reading switches, engaging solenoids and running the game logic. A protocol, (b) in Figure 3.1, allows external programs to control the flippers and buttons. The controller is described in detail in Section 3.1.

The only state given to the AI agent about the game being played is images recorded from an overhanging camera, shown in Figure 3.2. A vision system, (c) in Figure 3.1, has been designed to evaluate images quickly and discover the ball's location. Section 3.2 will elaborate on the specifics of the vision system.

Pinball is high speed and relies on physical hardware that can break. A physics simulator, (d) in Figure 3.1, assists with pre-computation of specific shot angles which eases the stress on the machine for learning and speeds up the vision system object detection. The simulator will be described in Section 3.3.

Once the ball is detected, the agent must make decisions in real-time. A modular AI framework, (e) in Figure 3.1, has been created to ease experimentation with different types of agents. The interface used will be discussed in Section 3.4.

3.1 Hardware

A hardware controller board created by University of Southern California researchers [29] is used to allow external operation of the flippers. This controller is polled from the computer via a USB connection. The controller maintains the local state of all motors, switches, and solenoids. Nathan Sturtevant created an event-driven system built on top of the USC

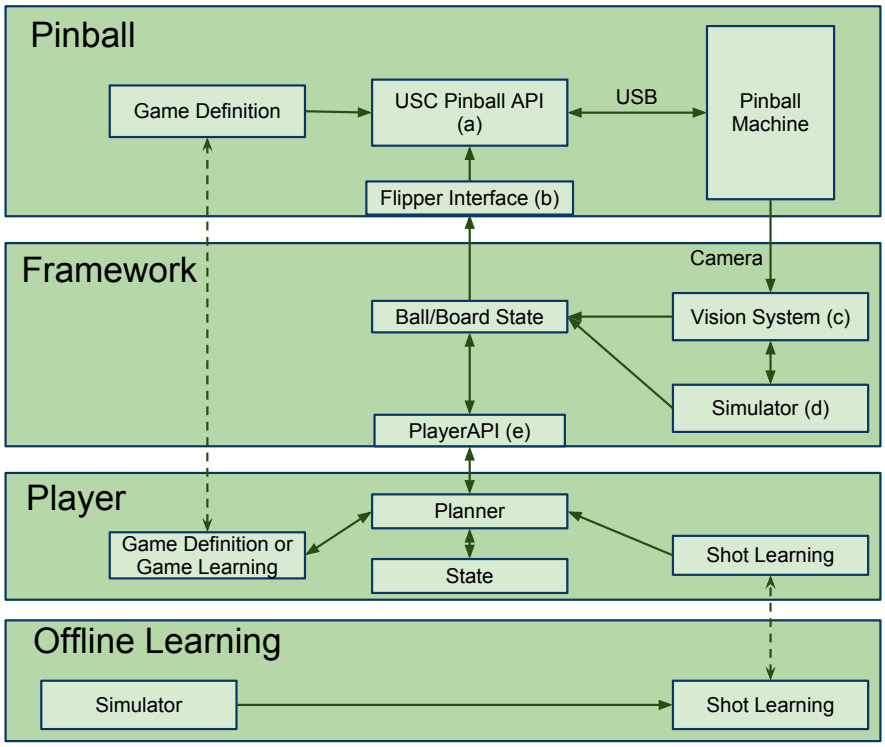

Figure 3.1: Outline of the Pinball AI project.

controller API. A game designer can use this event-based system and controller board to create their own game logic on the pinball machine, even creating a multi-player game using our multi-player server.

As part of this thesis work, the event-based system and controller library were optimized to increase the responsiveness of the hardware. The polling code was optimized which made it possible to increase the rate at which we can poll the hardware controller.

The optimization with the biggest effect was to reduce the time waiting for data. Polling of the switches was done sequentially and individually, rather than in one large chunk. Since the whole chunk of data is required before a new polling iteration can occur, we gain several benefits by doing a single large read. While waiting for data to read, the program is in a blocking state. While in this state, other processes on the computer can run while our process waits for data. When the process returns from blocking with the data ready to be read, we have all the data rather than the state of a single switch. There is overhead in making a read call to the USB controller and making one call is preferred to many.

Timer events, such as queuing solenoids and lamps, were centralized using the framework provided by the Simple DirectMedia Layer [23] library that handles key input and display for the controller. The performance improvement for timer centralization is minimal.

A small modification to the controller library inserted an external protocol for triggering

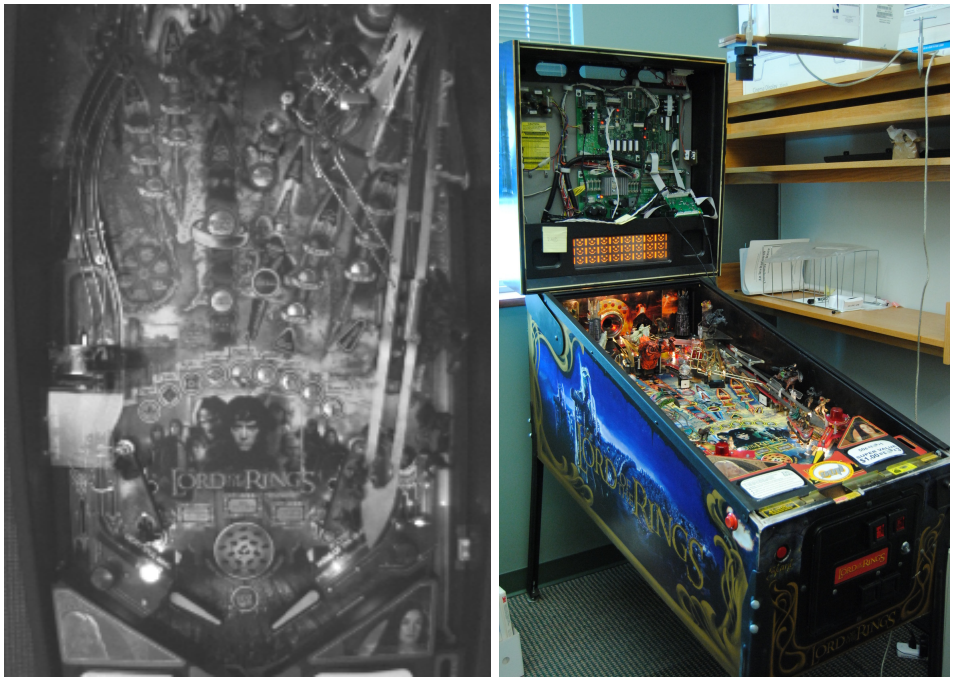

Figure 3.2: The view of the pinball play area from the overhanging camera and of the camera above the machine.

the player operable switches, such as the left and right flipper for an agent to use while playing. This protocol separates the work of the controller and the work of our pinball agent. This also prevents cheating by isolating a player from access to the state of switches and lights. The protocol is a simple UDP[1] packet with a 1 byte payload. This is a bit mask for specifying which user switches to engage or disengage. This allows for eight commands. For example, sending a packet with the one bit-value set will trigger the left flipper to fire and stay firing until a packet is sent with the bit-value two set. Likewise, the right flipper is triggered when the fourth bit-value is sent and disengaged when a packet with the eighth bit-value set is sent. Sending both the trigger and release bits for the same flipper will result in no action being taken on that flipper. There are five buttons on the machine, however, two are the coin return slots. If needed, the protocol could be extended to more than a single byte to include other commands. This network trigger system has a small overhead on top of a direct connection, however this latency is much less than the 1ms USB latency when run on the same machine.

This machine control protocol, shown as the flipper interface (b) in Figure 3.1, can be extended to a physical triggering mechanism put on top of a pinball machine. A much simpler board could be constructed to attach to the machine and push the buttons rather than a board that controls the whole game. In our system this would still have the same network and USB latency as the buttons need to be interpreted by the controller, but

[1]UDP is a connectionless network protocol that has low overhead but does not guarantee transmission.

with some additional mechanical latency. The advantage is that you could actually use this system on a pinball machine without any modifications and thus without any of the USB latency. However, then testing would have to occur on the original game that uses all the lights and has a limited number of lives or balls. The game we use to test disables all controllable lights and provides unlimited balls. The possibility of lights flashing is not handled by our vision system as described in the next Section.

3.2 Vision System

Ball detection and location is critical for an AI agent to be able to play the game. OpenCV [27] is a real-time computer vision library developed by Intel and released as open source. It is now supported by Willow Garage, a robotics research lab. OpenCV deals with the operating-system-provided camera drivers and provides a common front-end that is platform-independent. OpenCV also contains a collection of common algorithms that are useful for robotics and image processing. The algorithms that are useful for our vision system are background subtraction and object detection.

Pinball machines have a glass cover on top to contain the game and reduce noise. The glass covering with poor lighting conditions can turn some areas of the glass nearly opaque. With controlled lighting and being careful about reflections, the game could be played on the machine with the glass on. However, all of our testing and design has been on a machine with the glass cover removed.

The first camera we used was a FireFly 2 from Point Gray. This camera produced 640×480 pixel frames at a maximum of 30 FPS. The image quality was poor and grainy. To produce usable results the images needed to be scale down to 320×240 pixels to blend and reduce the visual noise. Working with the camera and building the framework highlighted that the camera speed is a very important issue. Pinball is a fast-paced game. The ball is able to move from one corner of the image to the other in a single frame at this camera's refresh rate of 33 milliseconds. This affected tracking and made it almost impossible to locate the ball reliably. This problem made it very difficult for an agent to play. We considered ways to work around the slow camera. One idea was to implement the trajectory analysis done by Johnson as described in Section 2.1. While this would help, it was not a proper solution. The more precision available to an agent, the higher quality the agent can be. At this point we began researching for a higher frame-rate camera.

Our research led us to use a Point Gray FFMV-03M2C camera, seen in Figure 3.3, that is capable of producing 122 frames per second (FPS) at 320×240 pixels. This camera produces crisp black and white images with very little static and does it very quickly. This camera is a big improvement over the FireFly 2. Not only were the images free of noise which made processing them easier, but it was four times faster at producing images.

Figure 3.3: A Point Gray FFMV-03M2C camera with a PELCO lens suspended over a pinball machine.

Our images are passed through many software layers before an AI agent is signaled to act. First the background of the image is removed. Then objects are discovered using the pixels that are in the foreground. The objects detected are then filtered using the simulator to leave only those objects that are physically possible from the last observed location. From this the ball's location can be observed and its velocity calculated. Then the AI agent is sent the location and velocity of the ball.

To get high performance we construct a pipeline known as the camera buffer. This idea is used in the RoboCup small size soccer league as discussed in Section 2.4. A computer is used to process the frames and execute the agent, potentially the same one that is interacting with the pinball game via the hardware controller. In our system, one thread in the computer process is simply tasked with retrieving the next image. This might also include filtering colors or converting the image to black and white. Once it has the next image ready, it pushes the image to the camera buffer pipeline which will signal all threads waiting for a new image. The display thread and the processing thread will be signaled. The display thread simply displays the image to a window on the computer screen. Optionally, it can wait until the processing thread completes to display a processed image with other data, like the current path of the ball. The processing thread will do background subtraction and object detection. These topics will be discussed in the following sections. After the processing is complete and an object is detected the AI agent will be called in the processing thread

with the latest position and velocity of the ball. A visualization of this pipeline is shown in Figure 3.4.

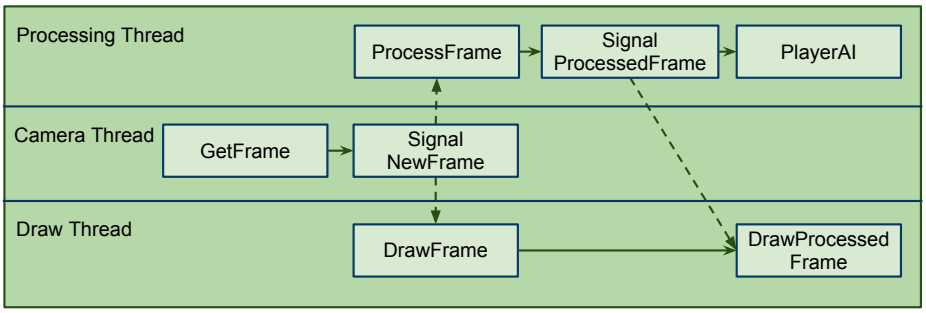

Figure 3.4: The camera buffer image pipeline.

3.2.1 Background Subtraction

Background subtraction is a key part of a vision system where you are interested in finding a moving object or an object that does not belong in the frame. It is often used in security applications to detect motion or intruders entering an area after hours. In pinball, we are interested in removing the detailed background such that the ball, which is not part of the background, will be visible.

The basics of background subtraction is comparing a frame to an image mask that represents the unaltered scene. There are many ways of accomplishing this task of building a mask, each with their own benefits and drawbacks. Method one, a static mask is an unchanging rule of what the image should look like. Method two, the Mixture of Gaussians [16], creates a probabilistic model for each pixel's intensity to create the mask image. Method three, a codebook [17], is an learned filter trained on an empty scene, constructed by measuring each pixel's intensity and how frequently it appears. Each of these methods will be discussed in the following sections. The end result of background subtraction can be seen in Figure 3.5. In this image we see the camera's view of the play area on the left and the white spot on the right representing a difference of the current frame from the known background. From the right image mask we can extract a location for the ball and proceed with object detection.

An example we will use to describe the background subtraction algorithms throughout this section is an outdoor scene. In this scene there is a tree with leaves and a breeze causing the leafs to rustle. The tree also has a few holes where the sky is visible through it. The gaps will shift and move as wind pushes the tree. Also in the scene, for a short amount of time, is a bird flying across the sky.

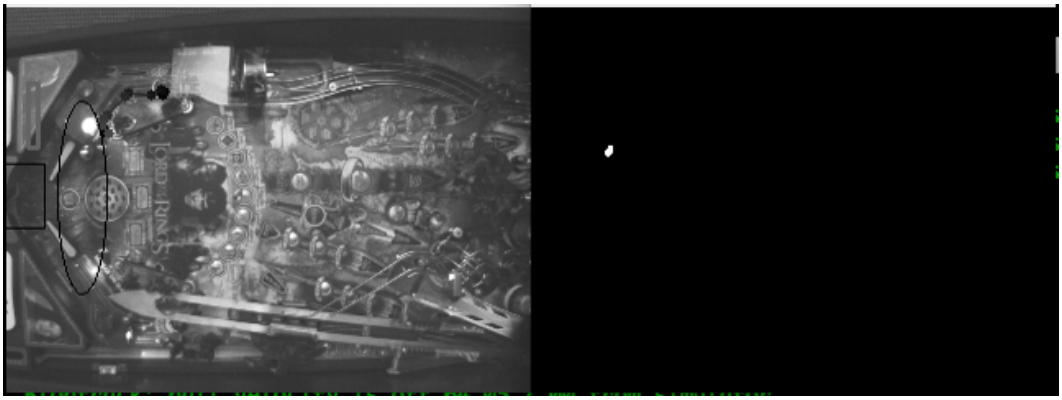

Figure 3.5: An example of effective background subtraction. The camera image is on the left and the foreground mask of the image is on the right. The square in the left image represents the area where the ball will be drained. The area where the flippers can interact with the ball is in the ellipsis in the left image.

Static Mask

The static mask is the fastest and most basic of the background subtraction algorithms, however it has many flaws. As the name describes, a static mask does not change. It is a frozen image of ideal conditions that is used to compare to the current image. Its main advantage is that it is simple to process a difference between two images. A subtraction of the intensity values of an image from the static mask results in an image where anything that is the same will be at removed leaving zero intensity or blackness.

Use of a static mask is effective over small time periods and in controlled areas. In our outdoor example, the wind pushing against the tree would cause much of the tree to be selected as being in the foreground. Changes in lighting, from sunlight or other sources, will cause the static image to be incorrect as well. The artifacts in the images, like random static or noise, will also show up as a difference in the image.

The game of pinball has large powerful solenoids. These solenoids can create vibrations in the machine and actually move it and its surroundings slightly. In Figure 3.6 the camera has been nudged slightly to simulate the vibration. This nudge causes much of the image to appear different from the mask. The movement of the machine means that using a static mask has to be eliminated as a possibility for background subtraction in pinball.

Mixture of Gaussian

Mixture of Gaussian (MOG) [16] is an online background segmentation algorithm. Each pixel has a small number of Gaussian distributions (usually three to five) that represent the color or intensity. Over time, these distributions model the scene and continue to learn. When a new image is checked against the model, any outlying pixel that does not fit inside that pixel's distributions is defined as being in the foreground. Together, the foreground

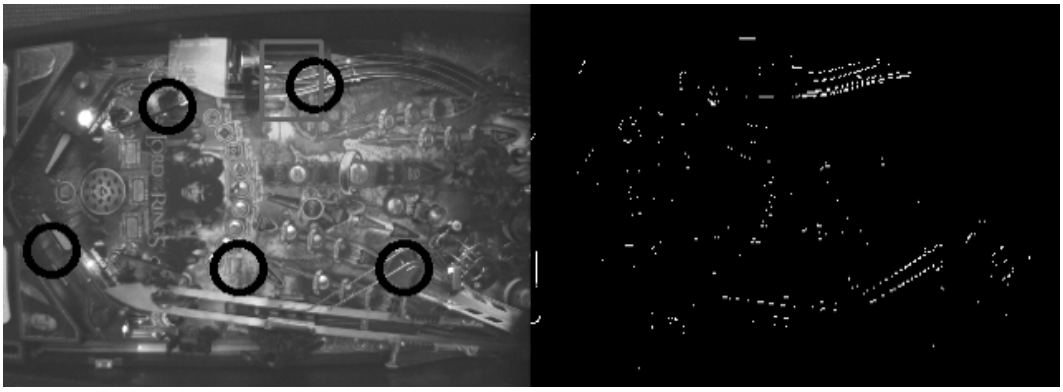

Figure 3.6: A small shake experienced by the camera will break a static mask approach. The mask will never adapt to these changes. Circles in the left image are false positives in the object detection system caused by the grainy foreground.

pixels define areas of the scene that have changed.

For each new frame, a pixel is compared to the set of Gaussian distributions. The Gaussian distributions are ordered based on their learned weights. A subset B of the distributions are selected such that the sum of weights are greater than a threshold T. If the pixel's intensity I is not within n standard deviations of any one of the Gaussian distributions (n is 3.5 standard deviations in our vision system), then the Gaussian distribution with the least probability for I is reset with a mean of I and a large variance. Otherwise, the first Gaussian distribution to come within n is updated. The update refreshes the weight, mean and variance of the distribution based on some parameters for learning rate. To extract the foreground out of an image, a pixel is marked as a foreground object if it does not match one of the Gaussian distributions in B. The pseudo code for MOG is in Figure 3.7.

MOG updates its model on each frame and converges to a new image mask. This constant learning makes it an excellent background subtraction algorithm for pinball and our outdoor scene example. The learning will also handle with slight bumps and movement of the pinball machine automatically.

OpenCV has implemented the MOG as part of its video module. The MOG algorithm is inherently parallel. Each pixel is computed and updated independently from the others. A patch was written for OpenCV which multi-threaded the execution of MOG by sectioning the image into horizontal slices. However, our solution on such a small image did not result in any speed-up. The speed-up experienced from threading MOG increases with the size of the images.

Codebook

A newer background subtraction algorithm is the codebook [17]. A codebook is a combination of ideas from both MOG and static image subtraction. The codebook collects pixel

Algorithm MOG()

Input: Image I

Output: Image Mask O

Globals: Gaussian Set G

Parameters: VarianceThreshold (vT), the learning rate (α),
Minimum Variance ($minV$), and Threshold (T)

foreach pixel i from I **do**
  $wsum = 0$
  $found = 0$
  foreach Gaussian g from G **do**
    $wsum \mathrel{+}= g.weight$
    if $g.weight < \epsilon$ **then**
      break
    $sqdiff = (i - g.\mu)^2$
    if $sqdiff < vT * g.Variance$ **then**
      $wsum \mathrel{-}= g.weight$
      $g.weight \mathrel{+}= \alpha * (1 - g.weight)$
      $g.\mu \mathrel{+}= \alpha * sqdiff$
      $g.Variance = max(g.Variance + \alpha * (sqdiff - g.Variance), minV)$
      sort G by $\frac{g.w}{\sqrt{g.Variance}}$
      $found = g.index$ // The Gaussian are sorted by weight
  if $found < 0$ **then**
    Replace weakest Gaussian g with a new Gaussian where $\mu = i$
  Rescale all weights for Gaussian g in G to sum to 1
  B = Set of Gaussians g ordered by highest weight to lowest where $sum(g.weight)$ is just $> T$
  if $found < |B|$ **then**
    $O[i] = 0$ // i was found in a highly weighted Gaussian
  else
    $O[i] = 1$ // i was not found by any of the Gaussian in B

Figure 3.7: Mixture of Gaussian Background Subtraction pseudo code.

intensities over a learning period. During the learning period it records the times it sees specific intensities to gauge if it is actually a part of the background. For each pixel during the learning phase, a codebook records a six-tuple for the minimum and maximum intensities, as well as timing information such as frequency of occurrence, longest time without an occurrence, and first and last access times. At the end of the learning period entries that have a low frequency are removed from the codebook. These are entries that really were not in the background but just happened to exist in the learning period for a little while.

In our outdoor scene example, the bird would ruin the mask of the static image subtraction method. The codebook method would record the pixel colors or intensities during the learning time. The codebook collects the different shading of the leaves as they move in the wind frequently. However, the bird has entries in the codebook across the sky as well. After the learning period is complete, a temporal filter clears entries where too much time has passed between occurrences. The bird is not likely to occur more than once per pixel. The typical temporal cut off is having recorded a second occurrence in at least half of the

training period time.

Codebook background subtraction is built into OpenCV and is available to our pinball vision system.

A disadvantage of the codebook method is after the learning period it no longer adapts. Thus, it suffers from some of the same problems as a static image mask. The vibration of the camera will never self-correct and the algorithm's effectiveness will be limited even if the change is very small. Codebook requires that your camera be absolutely stationary for the entire duration of the scene, which will not be the case in our pinball environment. The codebook algorithm will also fail to produce accurate masks when objects in the play field are moved. The *Lord of the Rings* pinball machine has an orange monster that can move itself. The training period will never see this occur and will not adapt.

Algorithm Comparison

In the previous sections we described two advanced background subtraction algorithms that work with our vision system and one algorithm that is not effective in the pinball domain.

Both the MOG and codebook algorithms are available as options. MOG has the advantage of being able to update as the game plays with no designated learning period. It can correct over time to provide an accurate background mask. The codebook method is slightly faster than MOG. Codebook does not have to update the model after the learning period has occurred. However, not adapting is a weakness of using the codebook.

Our vision system was run on a 2.4Ghz Intel Core 2 Duo with 4GB of RAM running Mac OSX 10.6.6. The MOG subtraction algorithm in our system that uses 3 Gaussian distributions per pixel averages 115FPS. This approaches the physical limit of our camera. The codebook subtracter can max out the camera frames per second at 122FPS but the mask will not adapt to changes past the learning time. The advantage of codebook over MOG is irrelevant as both subtraction methods can keep up to the camera. These subtraction algorithms produce slightly different mask images which can have a small influence in object detection but in general their performance is similar. We choose to provide both implementations to the agents, however the default is set to MOG subtraction since it is adaptive to changes with no significant impact on speed.

3.2.2 Object Detection

Once the background has been removed from the image you are left with a binary mask, with a white pixel representing a foreground pixel. Object detection is accomplished by taking these white pixels from the mask and grouping them together in a logical way to reveal whole objects.

Contour Detection

Contour extraction is built into OpenCV using Suzuki's connected component labeling algorithm [25]. The algorithm scans a binary image and sequentially assigns a group to a foreground pixel based on what other groups are connected to it. The algorithm proceeds from top to bottom and left to right. Suzuki's algorithm uses an extra one-dimensional table to store a union-find data structure. The union-find or disjoin-set data structure is a way to define group ownership quickly. A single update to the union-find table effects any pixels that belong to that group. Once a scan is completed in the forward direction, the image is then read bottom to top from right to left. During this reverse scan the group membership is updated. Alternating forward and reverse scans are repeated until membership does not change. In the authors experiments, this rarely needed more than three scans to completely label all the foreground pixels.

Contour extraction is able to reliably detect the ball, which looks like a white circle in the foreground mask. A more advanced detection algorithm that looks at the shape of the components could be used, but would be computationally more expensive. The basic contour extraction provided by OpenCV is all that is needed with a simulator running in the background. The simulator can be used as a substitute when the ball cannot be detected. This will be described in Section 3.3.

It is likely that many objects will be discovered in the scene. For example, a light flickering from the firing of a powerful solenoid has a strong probability of being identified as an object. The vision system must determine which one is the ball out of the objects found. The system filters out objects that are too small or big to be the ball. This covers many noise artifacts that occur, as well as a light flickering. Some of the lights that flicker shine onto the play area near the flippers. Filtering out locations, especially areas that the ball can be, can affect tracking the ball. The areas that the ball cannot be do not change and will be properly filtered out by one of the background subtraction algorithms. Once we have seen the ball, further detection becomes easier and faster using the simulator, which will be described in Section 3.3.

K-means Clustering

Another algorithm that can be applied to object detection is the K-means clustering algorithm [20]. The K-means algorithm is an expectation-maximizing algorithm. The goal of clustering is to group observations, in our case pixels of the foreground, into sets such that the sum of squares from the set's centroid is minimized. The K-means clustering problem is solved with a heuristic algorithm.

The algorithm for K-means clustering is described in Figure 3.8. Figure 3.9 demonstrates the first step of the K-means algorithm. An initial random distribution of means is created

to bootstrap the incremental clustering algorithm. We specify K, which is the number of groups to make. In our example where $K = 3$, the three squares in the first image are these random means. The next step is to associate all observations with the closest mean available, seen in the second image. After the association is made for all the observations, the means are then set to the current mean of the group of observations. The algorithm runs for n iterations or until it converges. This is a parameter called the termination criteria. Once the algorithm converges or terminates the means will represent the centroid of a group of observations. In our vision system, the foreground masks will have densely localized observations where the ball is. The foreground masks combined with the K-means clustering algorithm is a quick and accurate way to do object detection for pinball.

Algorithm kmeans()

Input: Observations O, number of groups k

Output: Tagged Observations O

Parameters: Termination Criteria C

foreach k **do**
    μ_k = random position in observation area
foreach Observation i from O **do**
    assign i to the group with the nearest centroid μ_k
while C is not met **do**
    foreach k **do**
        μ_k = centroid of observations i belonging to k
    foreach i in O **do**
        assign i to the group with the nearest centroid μ_k

Figure 3.8: Pseudo code for the K-means clustering algorithm.

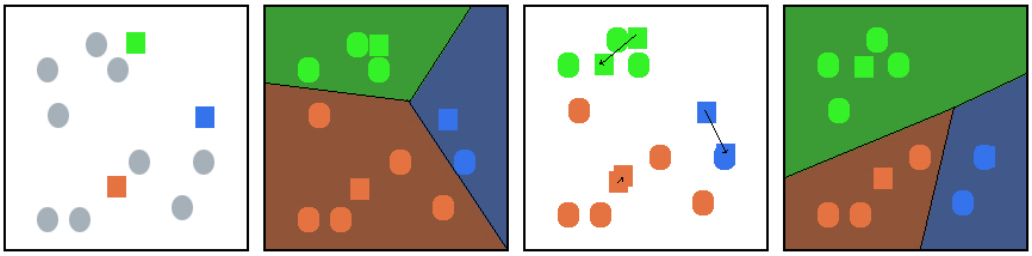

Figure 3.9: The first iteration of the K-means clustering algorithm. The circle is an observation and a square is the centroids of a group. The images from left to right show initialization, assignment of observations, calculating observation centroids, and the reassignment of observations.

3.2.3 Summary

Our vision system operates like a pipeline. Information is passed along the pipeline until the ball location and velocity is produced. First, the image is received from the camera and begins processing. The processing removes the background, leaving only pixels that change

from frame to frame. These pixels are then grouped together to form objects. Then, the objects are filtered based on size and location to reveal the object that is most likely to be the ball. Finally, the location is compared to the previous detections to calculate an estimated velocity. When it has finished with the current frame, the system starts processing the next frame.

3.3 Simulator

The key difference between video game and real world pinball is that machines can and will break down. During a game the parts of the machine experience wear and tear. A downtime can occur when a solenoid coil breaks or a fuse blows. Therefore the physical hardware is too fragile for most reinforcement or parameter learning algorithms. However, there still is a need to have some ability to predict motion on the board. To learn parts of the game offline a physics simulator would be useful. Ideally, one could pre-learn shot angles and the time to wait before firing.

3.3.1 First Attempt

A simple collision-based two-dimensional physics simulator was created. It supported circle and rectangle objects, as these could be used to model most of a pinball machine. A programmer creates a *Simulator* class object that can spawn all the objects that exist in the world. This simulator is simple and not generic or event based. It requires that the programmer iterate through objects to determine if a collision occurs after stepping the simulator. This is needed because the simulator did not manage the collisions; it was up to a creator of the simulated world to assign a new vector when a collision happened. This allows for inelastic collisions such as the slingshots. A slingshot launches the ball when hit rather than just collide and reflect.

The objects in the simulator were positions, sizes, and vectors. Each update of the simulator moved objects that were not fixed in place by applying forces. These forces were the previous two dimensional velocity and the experienced force of gravity on a 6.5 degree incline. The simulator did not model friction under the assumption that a pinball machine playing surface is slick and our object is a sphere which can roll.

The collision function was very simple. It was based on a circle-line distance function. A collision occurs when a line is less than the radius of a circle away from the normal vector of the line and the circle is between the two end points of the line. This function works well for long straight lines but becomes complicated when dealing with corners of objects. To speed up offline simulation, we tried to estimate the time until a collision. Instead of simulating the world in fixed time steps up to this collision, we jump to the moment before the collision would occur and update all the simulator object positions. This dynamic stepping did

increase the speed of simulation but the fine detail was lost.

The simulator is able to predict the resulting angle from firing the flipper at a specific time. The angle is stored in a table to be looked up during play. The table is indexed by the ball location and velocity, as well as the desired flipper angle wanted to make the right result angle. The array is 5 dimensions, however you can only index based on the 4 dimensions that were known in the table. Not all positions return valid result angles. For example the ball could miss the flipper all together. A binary search on the last dimension returned the flipper angle that provides the closest result angle. This is used to tell the time needed to fire. Since the flipper accelerates very quickly it was assumed that it would arrive at that angle instantly if fired.

This simulator is too simplistic. It fails to be realistic with specific areas of the play area, the side ramps in particular. The collision function was not generic enough to model the pinball machine's play area with any detail. It was time to look for other solutions for the simulator, observing the potential need for new features such as rotational momentum, friction, and real-time simulation.

3.3.2 Second Attempt

The features that are required to get a reasonable simulation are known. The shortcomings of the first simple simulator stem from the lack of advanced physics. There are many physics simulation engines available targeted for physics-based games. These engines are more generalized and have been debugged over time.

Box2D [6] is a highly portable physics simulator popular with web-based games. It offers continuous collision detection and a API very similar to our first simulator. A programmer can place objects in the world and call a step function. This allows for the reuse of much of the older simulator's shot angle learning framework. Aside from having a faster, accurate and stable physics engine, it also has collision callbacks and joint constraints.

These collision callbacks can be used to implement inelastic collisions. This is useful for simulating launchers, slingshots, bumpers and other objects that can create force. This is not necessary for online simulation, which will be discussed in the next section. With online simulation we can update the simulator with observations from the real-world. This keeps the simulation relevant and reduces the complexity of the simulation. The simulator acts as an observation filter which can be used for estimations.

To define objects in Box2D takes a few steps. First the object must be defined as a dynamic or static body. A static body allows an object to take root in the world and become unmovable by any force. We construct ramps and obstacles as static objects. The ball and flippers are defined as dynamic bodies. Dynamic bodies are affected by gravity and other forces. They are also declared as *bullets*, an object that undergoes a more complex

update to prevent it from warping or tunneling through objects when going at high speed.

Box2D implements continuous collision detection to prevent bodies from tunneling. The engine determines the first time of impact for a body when it is defined as a bullet. This is different from other bodies with basic collision detection that update the position and simply check for overlapping shapes. Continuous collision detection in Box2D is implemented by sweeping from the current position to the position of the body if no collision were to occur. During the sweep, the body iterates through all other objects until a minimum time of impact is determined. If a time of impact is found, then the bullet body's position is updated to the position at that time. Continuous collision detection generates a great amount of detail from the simulation, but has a great cost in performance. Not every object should be defined as a bullet as this will be slow to process. However, in pinball, the ball and flippers move at a speed that will have tunneling problems. Therefore we must take the performance hit to ensure accuracy.

Once a body is initialized with its body type we can specify its initial position and velocity and assign it to a shape. Box2D supports circles, polygons, and a generic shape which can be extended to implement new shapes. Other physical properties can be assigned along with shape. The world creator can change the density and friction of world objects. Once all this is set, the objects will exist in the simulated world. By stepping an amount of time the world will update as if time had passed.

More complex physics interactions are possible in Box2D. Joints will connect objects together and enforce restrictions. A *weld* joint acts to connect to objects at a specific point. The *revolute* joint sets a relationship of one object being able to rotate around another. The revolute joint can also apply forces like a motor, allowing it to spin another object. Complex objects can be created by combining joints and bodies. For example, we construct the pinball flippers using a weld joint on the pivot point and then connect a revolute joint from that to a flipper body. We specify on the joint that the flipper can only move from zero to sixty degrees like the LOTR pinball machine. We then set up functions to apply torque to the revolute joint to simulate a player firing the flipper.

Using bodies and joints, a basic model of the LOTR pinball machine lower playing area was constructed in a Box2D world. The world in the simulator is made up of a ball, two of the flippers as discussed previously, and the static bodies that represent the area around the flippers. This simulated world is an accurate representation of the physical area of the LOTR machine. The slingshots are not implemented as they have a random element and are not necessary to hit the ball.

The shot angle table from the first simulator, described in the previous section, was reconstructed. This time the table was populated using the new simulator. The new simulator was not only faster and more accurate, it now included the side ramps, a key part of the

play area.

The table was constructed twice to investigate the effects of rotational momentum. The outward angles recorded from the simulator were the same when rotational momentum on the ball was disabled. This helps to verify a previous assumption that rotational momentum is not worth the extra computational complexity.

The quality of the simulation is good. Box2D is designed to be quick and real-time for use in games and it provides the accuracy needed for our pinball simulation to work with the actual machine. The library is mature and well maintained. There are other libraries available if more accuracy is required in the future. Bullet Physics Library [9] is used in commercial video games and special effects in movies. This physics engine supports three-dimensional simulation and is capable of reading in worlds that have been created using 3D modeling software. Box2D is limited to the x-y plane and simulation details can be lost. However, the current vision system is not capable of determining the depth of the ball either. We have assumed that the extra dimension is not critical to playing the game, as the ball rolls on the play area most of the time.

3.3.3 Simulator Filtering

The new simulator based on Box2D is fast, as it is very basic. Using it the system can approximate where the ball should be, online, while the game is played. On each new frame, the simulator is stepped by the time since the last frame. The ball's location and velocity is updated inside the simulator world.

Using the physical knowledge we can create a region of interest in the recorded image. Our region is centered at the ball's potential location and extends out twice the estimated velocity in each cardinal direction. We should only care about objects found inside of the region. If the ball is not detected here we can extend the search area. This speeds up detection by only dealing with objects in a smaller area. However, this speed-up is only fractions of a millisecond and can only be achieved when there are multiple detections in the play field. The filtering allows the vision system to eliminate potential objects earlier. The ball's location and velocity in the simulator is updated when a ball is located. This keeps the simulator current in case there is a small inaccuracy in the physics. However, if no ball is discovered in the region or image we could pass on the simulator's approximation of the ball location and velocity to an AI agent.

3.4 AI Framework

With a vision system that can track the ball quickly it is time for action. Players of the pinball game will know the position and velocity of the ball. Using this state knowledge, they can interact with the game by using the flippers. In the future, the current score should

```
typedef struct {
    int x;
    int y;
    float vx;
    float vy;
}Ball;

class PlayerAPI {
        void queueFlipper(int flipper, double time = 0.0,
                          double span = 0.25);
        void launchBall(double time = 0.0);
}

class Player {
        virtual void update(PlayerAPI *api, Ball *b) = 0;
}
```

Figure 3.10: The AI agent interface.

be sent to the player along with the ball's information. A common method and protocol for scoring should be added to the USC controller library that will work with any games that are created using it.

Our agent's framework is based on an abstract C++ class. The agent implements a method that is called when a new ball location is discovered. This method will have the ball location as a parameter. See Figure 3.10 for the details. The PlayerAPI class provides the agent with access to the flippers and ball launcher. An agent can queue up flippers to fire when anticipating shots based on trajectory. This pinball AI strategy of calculating the time until impact was employed in virtual pinball agents, as mentioned in Section 2.1 and Section 3.3, so the ability to queue shots was added to the API.

3.4.1 Novice Player

A novice player of pinball does not care about outgoing trajectories of the ball. If they see the ball around the flippers they just try and hit it. Our novice player does this. It monitors areas around the flippers. If the ball enters or passes through them it attempts to fire a flipper and hopes to hit the ball.

This AI agent is built upon the idea of zones. If a ball enters a zone the agent would want to react somehow. A Zone class provided in the framework contains an abstract callback that can be overloaded. This Zone class works with line segments to determine if the ball passed through a zone, even if the current location is not inside the zone. This prevents a ball that is moving quickly from warping through a zone without executing the callback. For rectangles we use the Cyrus-Beck line clipping algorithm [10] to accomplish this. The Cyrus-Beck algorithm checks the position on the line in which it intersects with an edge of

a zone as though the edge was an infinite line. If this intersection position is between the two points of the line, then the line is inside the zone.

This agent is able to hit the ball and stay alive. More details of its effectiveness at playing pinball will be discussed in Chapter 4.

3.4.2 Advanced Players

To reach a higher level of play a more sophisticated planner will be required. The progress for an advanced AI agent for pinball will likely follow the same skill levels as described in Section 2.1.

Not all shots are equal. The goal of survival is important but will likely not be the way to achieve high scores. The need to hit specific targets occurs often in pinball games, and the benefit is receiving a large bonus. A more advanced player would need to know these targets ahead of time or learn that they exist. Most targets typically get a lamp below them lit up to mark that they are a current target. However, lamp use is a limitation in our current vision system implementation. Another way this could be implemented is with the construction of a target declaration protocol. The targets would be sent from the USC library and the agent would know what places the ball needs to be. The target protocol could be used as a stepping stone to the more advanced vision system.

Knowing what targets to hit, the agents would then need to learn to control the ball. Having control would involve learning to cradle the ball and even learning when it is appropriate to attempt to cradle the ball. The simulator determined if the ball was able to be cradled in the flipper when generating the shot angle table off-line. A ball was considered cradled when the ball was not launched off the flipper into the play area and remained in the flipper for a second without moving. However, the cradling results did not get verified on the physical pinball machine.

Once an agent can cradle the ball successfully, precise shots can be attempted from a more stable starting point. Along with cradling, the ability to pass the ball from one flipper paddle to another is important to targeting both sides of the play area. After that, the human experts learn more tricks that offer them even more control.

Having the ability to hit specific targets when needed and knowing where to shoot the ball at the time are the big steps toward a pinball wizard AI.

3.5 Network Multi-player

As part of the AI framework we built a client-server multi-player interface for pinball games. This interface not only allows new game types for pinball to be created but also a way to compare AI agents competitively. In this section, we describe what is required to support the new extension to the classic game of pinball.

Game styles for parallel multi-player pinball have not been approached before. Some examples of the possibilities for game types are cooperative, score attacks, adversarial games or a combinations of these. Cooperative games have the players working towards a common goal. Together they must demonstrate control and planning to win. A score attack game is similar to playing single player pinball, however, you can see the score of your opponent during the game and after a set time limit it determines a winner based on score. Adversarial games have a player directly effect what the opponent must do. For instance, if one player hits a specific switch then the opponent must try to hit that same switch within ten seconds or the first player will be given a score bonus. There are numerous other game ideas that can be created.

There is also room for asymmetrical difficulty or play in multi-player pinball. Follow the leader game types, where one person must try to mimic moves of another, is an asymmetrical game. It is more difficult to have to hit specific targets than it is to simply keep a ball alive. This can allow pinball wizards to enjoy playing with novices in the same game.

In building a multi-player server, information has to be communicated between the players. There are two considerations that determine the amount of information that needs to be exchanged. Pinball games can become quite complex, however, the game state is not changing drastically over time. The network pinball server must also allow for quite different games and must not be specialized for any particular game type. The best way to achieve this would be to leave the server as generic as possible. Our pinball server is a distributed key-value dictionary that contains the current state of the game. This is similar to two people sharing a blackboard. Both parties can see what the other writes and they can modify anything on the board as long as they know where it is.

A game developer using our network library simply creates C++ variables of our *NetworkVar* class with a unique string key and type. Standard *get* and *set* accessors exist for this class. However, when a variable is *set*, the server will be notified of its new value and all other clients playing the game will be sent an update for their local values. Using the *NetworkVar* class, the networking code is abstracted away from the programmer.

The protocol for this network server is based on Google's protocol buffers library (Protobuf) [14]. Protobuf is a open source serialization library for C++ and Java and is used internally at Google. Protobuf serializes classes simply and provides a way to design protocols quickly.

The Variable Server (*VarServer*) has a simple handshake with the clients to negotiate a unique player id that the client can send updates with. The client sends the name it would like to be identified with and the server will return a integer identifier that is to be used for the rest of the connection. Identifiers are given out to the clients incrementally. After the client has a valid player identification number it can create and modify variables

on the server. The client sends a initialization message to the server describing the name, type, and a client specific identifier for this variable request. When the server responds to this request it provides a new global identifier number for this variable, at which point the request identifier is no longer needed. It is possible that this variable already exists on the server if it was initialized previously by another client. If the variable exists, the client is sent the current value of the variable along with its global identifier number. After the player has the global id for the variable it can send updates to the variable by sending their player id, variable global id, and the new value or data to store in the dictionary.

The server communication and synchronization is hidden behind the *NetworkVar* class. Game designers can use the poll-based method of accessing variables or they can set a function to call when the variable is updated. The designer does not need to be familiar with network sockets in order to make a networked game with the library.

During a test of multi-player pinball we discovered a few flaws in our pinball game and protocol. The machine has some switches that will be triggered twice when entered. The game that was created had many switches, such that when triggered would toggle a variable. By hitting a switch that gets triggered twice the state would toggle, then toggle back and remain the same. This is not a problem with the protocol but a problem with the game design. To make these double switches act like single switches they must be treated specially with a timeout. The timeout would be enforced by the game logic to make sure that only the first trigger is reported.

Another problem we faced was introduced by network latency. Currently the client-server model used assumes that when you write the state there is zero latency to the server. In the time it takes for the update message to reach the server the state could have been changed already. This is not reflected back to the client and the logic is assumed to be correct. In our game, instead of writing a special value to toggle the current value, a binary value is written. A case came up in the game where the value was quickly toggled twice but the opponent attempted to toggle the value in the middle of these two changes. One player made the value change from false to true and back to false. The other player saw the value change to true then hit the switch and made the value change to false. Both players assumed they were setting the value false, however this affects scoring in the game as both cannot set it false.

The desyncronization problem in pinball multi-player is unique in the world of multi-player gaming because it is a physical object playing the game. In video games the program has the ability to render, modify the game world, and change previous interactions as needed to mediate latency. In pinball the ball is a physical object moving around that the game has no immediate control over. The game cannot slow down or stop the ball to determine what state the server thinks the game should be in.

A simple idea is to include execution timestamps in every transmission. The server then could buffer and organize transmissions by the timestamps and execute them when it is their time. However, this creates a much larger problem when the client's clock on the machine is different than the server's or the other clients. This is a problem for computers doing financial transactions inside a bank. The transactions cannot have variable times depending on what machine a client is using, just like game logic cannot be determined by who has the computer clock that is different. We did not implement timestamps in our multi-player server.

The Network Time Protocol (NTP) [21] was designed to synchronize the clocks of computers. NTP is able to keep computer clocks within milliseconds of each other. It took nine years of work and refinement to achieve that accuracy. NTP is an Internet service that is an organized group of servers designed to exchange timestamps and can be used to adjust clock frequency to maintain accuracy. A few of these servers determine the time from external sources. The time sources may be expensive atomic clocks or just radio broadcasts of timecodes produced from those clocks. The rest of the servers will synchronize the time based off these few servers in a hierarchical fashion. Those that are closest to the timing hardware are considered more accurate than those on the fringes of the network.

Keeping the clocks in sync does not solve the whole problem faced in multi-player pinball. There is still a question of how far do you have to set the execution time in the future to ensure fairness for all players and still have the game feel responsive.

In industry, many multi-player video games use the idea of turns or ticks; taking snapshots of the state at a regular interval to be shared with other players through a host. In *Age of Empires* [4], actions to be performed are bundled into turns by time (200ms). A turn is scheduled to be executed two turns in the future and only when all clients have all the other clients actions to execute. This creates a 600ms delay before an action would actually be executed, but that all clients would execute together. If a player is suffering from network problems the game would slow down while waiting for the turns to complete, thus ensuring all clients could proceed together. Each player's game simulates the same actions of all the players, in the same turns. In pinball, the game cannot slow down the ball or the game logic. The machine would look unresponsive if hitting a switch did not change the state because it is waiting for actions of a previous turn to execute.

The networking paradigm in the source engine from Valve Software [8] is partly what inspired the *NetworkVar*. The state of variables in the client are linked to the server's instance of that variable. The main difference between implementations is that the source engine client cannot directly change the variables, but only see changes to the variables based on their actions. The source engine ticks are much faster than *Age of Empires* (33ms) due to the faster pace of the games. However, the server only sends broadcasts every 50

milliseconds giving the server time to buffer late inputs into the next broadcast. In the source engine, the client only sends the user input to the server. The client will render the current state of the world based on the local networked variables. The local network variables are updated by the regular broadcasts from the server. To make the game look smooth the client can also perform interpolation on values, in the case that the transmissions to the server were lost and are in the process of retransmission. It is entirely up to the server to act out the client's input and update the state of variables for all other players. This requires that the server has absolute knowledge of the game being played.

To implement ticks on the pinball API, the game server would contain all the pinball game's logic. The client would send packets to the server containing which switches were triggered and in what tick. The server would then buffer these throughout its tick, setting the state of the game, and send regular state updates to the clients or instruct the clients which lights to turn on and what to show on the display. The *VarServer* was designed as a low bandwidth generic server for specific clients. Communication between server and client was done only when necessary. To fix the desynchronization present in the current implementation, the multi-player system would have a specific server that broadcasts regularly with more generic clients. The server sending isochronous broadcasts creates a clock that is unique and shared with the clients and not any particular machine's internal time.

Once games are designed and the protocol is made more stable, a man versus machine tournament could be held to compare an agent's programming to a real pinball wizard in a situation where they are able to affect each other's play. An example game that could be played at a man-versus-machine pinball tournament is follow the leader. In this game, a player is designated the leader and the other is the follower. As the leader makes certain shots, the follower is told to make those same shots, in the same order. It requires a lot of skill and some luck to be able to follow. An impressive display for an AI agent would be it following and keeping up to a leading pinball wizard who is repeatedly making skillful shots. If the players took turns leading and following this would be a great comparison for the skill level of an agent.

3.6 Summary

A robot is able to play pinball. The robot can see and track the ball and even attempt to hit it with some success. Using a hardware controller the game is opened to external interaction. With background subtraction the vision system can see what is changing. The system can then find a ball and tell an agent where it is. The novice agent will strike the flippers when it thinks it will hit the ball. There is still much to do to become a real pinball wizard. Taking control of the ball will be an important next step, as will knowing where to shoot the ball and how to get it there.

Chapter 4

Experiments

The framework for pinball AI allows us to measure performance of the player. There are many different ways to categorize the performance of a player. The time a ball is kept alive can be a demonstration of reflexes and control. The player's score is an overall judge of how well the player is doing or how well they can exploit the game. A more advanced skill is the ability to complete quests in the pinball game. Quests can be a series of targets to hit in a limited time or hitting a subset of switches without losing the ball down the drain.

Currently the only recorded metric used is the time the ball remains in play. This metric is also known as the survival goal. The experiments focus on the ability to track the ball and hit it. More sophisticated goals would involve scoring but the novice AI lacks a planner and a way to select targets to achieve high scores. High scores here are mostly the result of good luck. This is similar to a novice human that simply wishes to play longer, unaware of how the specific switches will change the game or affect their score.

4.1 Experiment Design

The survival goal is easy to measure. The time since the ball was launched is recorded when the ball is drained in the bottom of the pinball machine. The start and end state are the same and another trial can be run immediately. This allows for automatic testing with little human interaction.

The game used for testing was created using USC's pinball API library and the event library built on that. The game has two modes: remote control and human. The start button toggles between them. In remote mode, the flipper buttons are disabled and the pinball machine responds to the computer flipper protocol. However in human mode, the game plays the same but without the flipper protocol active. While playing the display shows the amount of time the ball is alive. When a ball is drained, the system updates the display with the average ball life time and population variance and then launches a new ball. The same statistics are reported on the computer running the pinball game and they

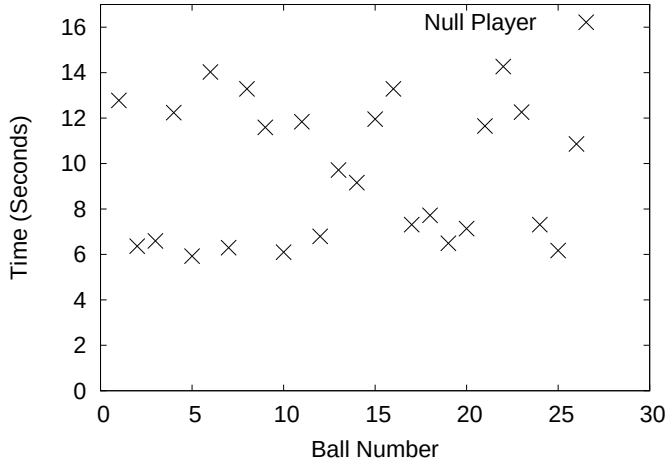

Figure 4.1: A baseline for ball life time when doing nothing. This is the minimum life time any player can get regardless of skill.

are recorded.

To prevent players from cradling the ball forever, the flippers can only be held in the engaged position for a maximum of half a second. The flipper can be fired again immediately after the flipper is disengaged. This restriction is implemented in the game by lowering the flipper after half a second without the player or AI releasing the flipper. This time limit keeps the ball moving for the length of time.

This game can be played by both humans and AI, and allows us to determine the relative performance of the AI.

4.2 Survival Experiment Results

To determine if the AI performs well we ran many trials. Each trial consists of twenty six balls or lives of the survival game. The trials were played by three different players: the null player, a human novice player, and the novice AI player. The null player is a special player that does nothing but launch the ball. The Human player (Adam Metcalf) played one extra trial as a practice round, playing 2 trials in total. The novice AI player is the agent described in Section 3.4.1. There is no spastic or random player to help reduce the wear on the machine.

The time a ball is alive without any interaction is an interesting statistic. The null player's times represent the minimum amount of time that it is expected a ball to last. Figure 4.1 shows that even doing nothing has some variance in it. After 26 balls, the average life time for a ball was 9.58 seconds. The standard deviation for the trial was 2.96

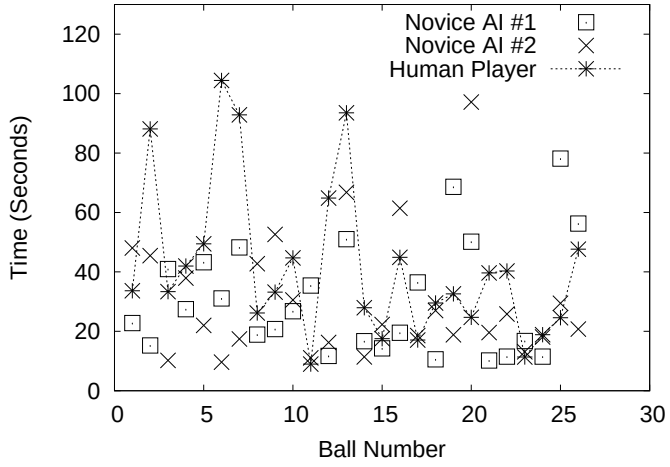

Figure 4.2: The novice human's survival along with the Novice AI's results. Times measured and exported by the survival custom pinball game.

seconds. A ball that is alive for approximately ten seconds is about the same as the player doing nothing. This gives us a lowest possible value to compare to.

The human player was not used to playing the machine and was given a 26 ball practice trial. The first practice trial ball's averaged 26.69 seconds per ball. The standard deviation of the life time was 21.76 seconds.

The second trial for the human player took place after a brief rest. The average for the second test trial is 42.00 seconds with a standard deviation of 26.23 seconds. The time did increase between the practice trial and the test trial. For a human, practice will help to improve your ability to keep a ball alive in pinball. The results of the human test trial are listed in Figure 4.2. This shows a randomness in the ball's life time. This is mostly due to the player's inexperience and reliance on luck.

It is interesting to observe the effects of human stamina in pinball. The values of the human trials appear to have a decreasing trend after so many balls. Splitting the human trial into two groups of 13 balls at the midpoint shows that they are from different means and are statistically significant at α=0.05. Fatigue from being constantly alert and forced to have quick reactions affects the player's ability to keep the ball alive. The experts of pinball must not only have tricks to control the ball but it appears that they might possess the ability to keep their reaction times up and stable over longer periods of time. The AI framework does not have to worry about stamina and so it will have an advantage over longer periods of time.

Our novice AI described in Section 3.4.1 also played the survival game. The results from

the AI's first and second trials of the survival game are in Figure 4.2. The novice AI's first trial (Novice AI #1) averaged 30.54 seconds per ball with a 19.01 second standard deviation. The results are very close to the human trials. The AI is capable of playing at a level similar to a novice human when attempting to keep the ball alive.

Overall, the agent performed as one would expect a newcomer to the game. The agent would detect that it could hit the ball and fire a flipper. This is not how experts play the game. The more advanced players prefer to be in control at every moment possible. The restriction on the time a flipper can be engaged does limit the amount of control the player can have.

A few interesting scenarios came out of the novice AI's play that was not intended. The agent was able to pass a ball from one flipper to the other to be hit. This came about by the agent firing the flipper too soon and the ball rolling off the end of a fired flipper. The agent would then retract the flipper that could no longer hit the ball and upon seeing that the ball could be hit by the other flipper, it was fired. The passing move is one of the more advanced skills that a stronger AI could posses. In this case, the novice AI was just lucky. The demonstration of control was not actually planned. Another skillful shot the novice AI was able to perform without intention is called the flip trap. This shot is performed when a ball is falling toward a flipper and the player would like to cradle the ball. The agent fires a flipper such that the flipper makes contact with the ball at the moment before it is full extended. The remaining force in the flipper cancels out the downward momentum, essentially stopping the ball on the flipper.

A few balls from both the human and AI show numbers around ten to twenty seconds. The results in Figures 4.1 highlight this. These times represent moments where the player had either missed the ball on the first hit or was "unlucky" on the first couple of hits. Getting unlucky on a shot might involve having a slow reaction and hitting the ball with the flipper in such a way that it will drain. To a novice it is nearly impossible to recover the ball from these areas. More advanced players will learn to work around and avoid these shots all together or just be able to react in time.

To demonstrate the significance of reaction time, another two trials of the Novice AI were run. One trial had the frame rate of the vision system limited to 40FPS. Figure 4.3 shows the performance of the AI agents who differ only by reaction time. With the lower reaction time the agent is still capable of playing the game. In fact, the agent did surprisingly well on some lives but was not as capable as the normal agent. The average life time of the ball with the 40FPS agent was 25.1 seconds with a standard deviation of 17 seconds. The quicker agent (120FPS) averaged 30.55 seconds per ball with a standard deviation of 21 seconds. The results of the regular Novice AI's second trial were consistent with its first trial.

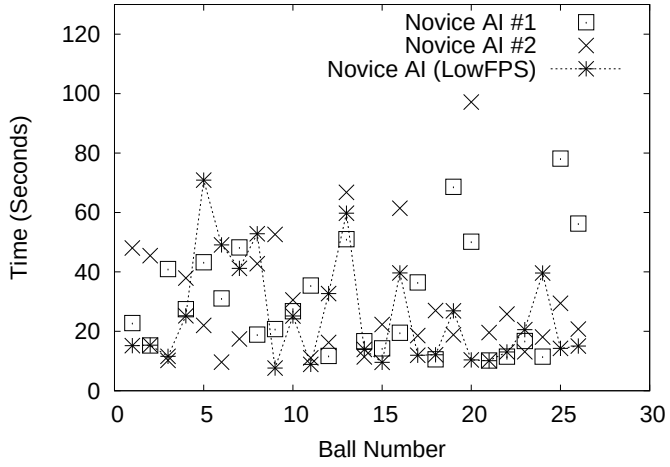

Figure 4.3: The Novice AI's performance based on frame rate of the vision system. Times recorded by the survival custom pinball game.

Player	Mean	Std. Dev.
Null	9.58	2.96
Human Practice	26.69	21.76
Human Trial	42.00	26.23
Novice AI Trial #1	30.54	19.01
Novice AI Trial #2	30.55	21.01
Novice AI (LowFPS)	25.10	17.73

Table 4.1: Life time results of the different 26 ball trials.

One-way ANOVA F-tests were run on the data sets to demonstrate statistical significance using α =0.05. From the tests, the mean of the human and the low frame rate novice AI are different. However, the same cannot be said for the regular Novice AI and the human. The results were not different enough to prove significance. All players, agents and human, were significantly better than the null player, and thus are better than doing nothing. The splits of the human trial, that were used to check the effects of fatigue, were tested against the novice AI. The first 13 balls of the human trial were significantly better than the novice AI. This is something that could not be said about the whole human trial because of the weaker second half.

These results, displayed in Figure 4.2, are promising for the future of pinball AI. The novice AI is not playing with the ability to learn and it is capable of playing with only a small amount of knowledge. In particular, the agent only knows where the flipper area is and it receives updates to where the ball is and its current speed. With only that little bit of information, it is able to keep the ball alive for a reasonable amount of time.

4.3 Framework Performance

During the experiments we determined that the system using MOG and K-means clustering averages 115FPS. This provides a 8ms frame refresh rate. With a small amount of time to send commands to the board, which is less than 16ms, this means that in the worst case the framework has a 24ms reaction time. The average vision-based reaction time according to Human Benchmark [3] is currently 215ms with the fastest official times being 100ms. The lower frame rate novice AI operating at 40FPS has a worst case reaction time of 41ms, which is still faster than that of a human.

The difference in reaction time between a human and the AI framework is very large. The human players can make up for the slow reaction time by predicting the location of the ball and timing their shots ahead. The Novice AI does not require that much planning to achieve the same results. In situations where players cannot foresee the motion of the ball, the AI will have an advantage by being able to react quicker.

A single mini-trial of ten balls was run in a version of the game where the lamps on the machine were able to toggle. Every half a second a random lamp on the machine would turn on and the previous lamp would turn off. The online simulator filtering helped a small amount to make sure that the ball was the object passed on to the agent but it did not work all the time. The average life time of the ten balls with flashing lights was 21.06 seconds with a standard deviation of 12.74 seconds. The agent was able to hit the ball sometimes but had a few trials where it was not able to hit the ball at all. The framework as it is, is not capable of playing with lamps effectively.

A small targeting test showcased the ability an AI player could have when the ball is in a known and static state. The ball was positioned into a engaged flipper such that it is cradling the ball. The machine would then disengage the flipper and fire the ball at one of five targets. The machine was given ten attempts at each of the targets. The five targets chosen are highlighted in Figure 4.4. The names of the targets from left to right are: Left Ramp, Center Ramp, Palantir, Right Ramp, Right Orbit.

The machine, shooting from a cradling position, was successful in hitting a designated target in 25 out of the 50 attempts. A human (Adam Metcalf) attempted to hit the same five targets as the machine and was successful for 18 out of the 50 attempts. For some targets the accuracy is as good as 70%. Targets farther away from the center appeared to be harder to hit both for the human and the machine. The detailed results from this experiment are listed in Table 4.2.

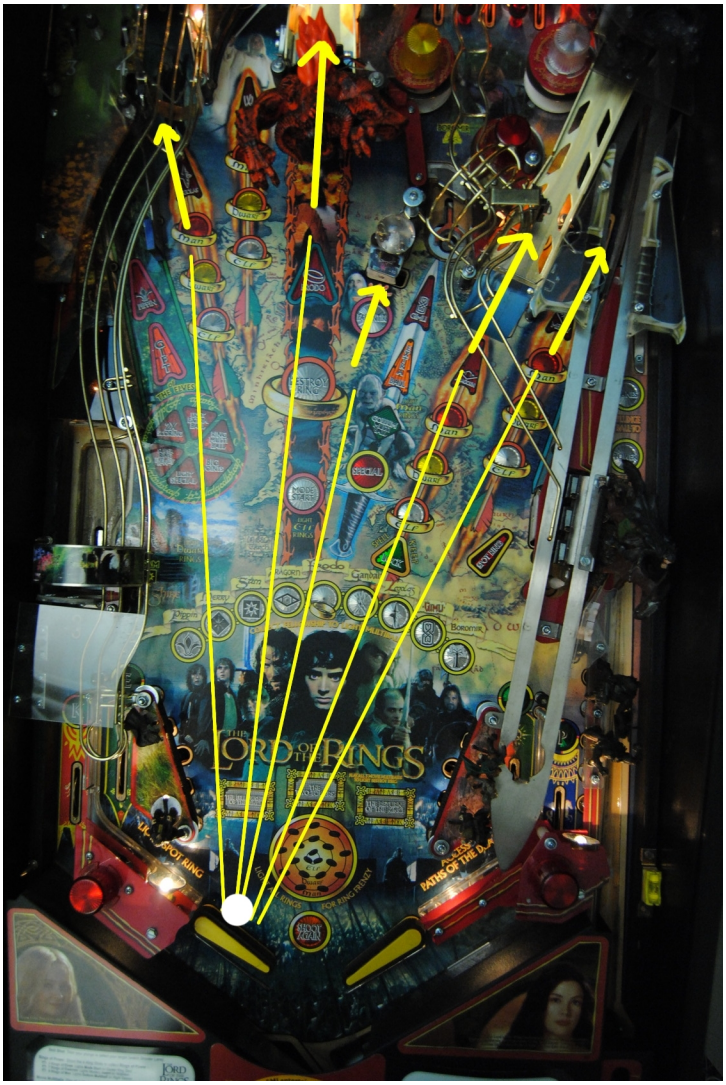

Figure 4.4: The five targets used for the targeting experiment.

4.4 Conclusions

The AI framework using the novice AI is capable of tracking the ball and reacting well enough to perform similar to a novice human. With more logic behind the agent's actions and combined with its quick reaction times, it should be able to outperform a human at keeping the ball alive. An AI will not suffer from the fatigue that was seen in the human trial.

Target	Machine	Human
Left Ramp	4 out of 10	2 out of 10
Center Ramp	7 out of 10	4 out of 10
Palantir	7 out of 10	6 out of 10
Right Ramp	6 out of 10	3 out of 10
Right Orbit	1 out of 10	3 out of 10

Table 4.2: Detailed results of the targeting experiment.

Chapter 5

Conclusion

The pinball domain using actual pinball machines has been opened up for researchers. Pinball is an older game where tricks, reactions, and planning have a impact in a player's ability to play.

A vision system combined with USC's pinball game library allows software agents to play a pinball game. A high frame rate camera provides clear images that are then processed to reveal the location of the ball. Once the ball is found, AI agents can take action to play a game.

The reactive novice AI demonstrates that the framework can compete with a beginner human player at simple tasks. At times the novice player looks as though it is doing something intelligent. One might confuse intelligence with skill when it is showing its quick reaction time. There is much to be done to become an AI pinball wizard but a foundation to proceed now exists.

5.1 Future Work

Since pinball AI using a real machine is a new idea, there is a lot to work on. However, there are two directions to continue researching: using a physical machine or using a simulator.

To put the focus on new AI functions, many game domains have simulators that are used in competitions to reduce the cost of entry. Not every lab interested in pinball AI can afford to purchase a pinball machine and equip it with USC's controller and a high frame rate camera. A way to proceed in this direction will be discussed in Section 5.1.1

The other way to proceed is to continue to improve the current architecture. This can be improvements to the agent's logic, as described in Section 5.1.2, or improvement to the underlying framework, which will be explored in Section 5.1.3.

5.1.1 PinMame Simulator

The *Visual PinMAME* program, discussed in Section 1.1, is a great program for running real pinball games in an emulator. This program takes the game logic and brings them into software. Currently *Visual Pinball* connects to the emulated machine to play the games. While connected, it displays the world and simulates the physics of the pinball machine. While simulating the physics, it will also trigger switches which are connected to the emulated machine to react as if the real game was being played.

Visual PinMAME and *Visual Pinball* are only available for the Windows operating system, but both have been open sourced for non-commercial use. The binary dumps of commercial games (like *Lord of the Rings*) used by *Visual PinMAME* are not free to use without owning the machine, but there are free games available. The combination of these two programs seem like a good start for a simulator world for AI. If the AI framework described in this thesis is transfered to this system it could allow anyone to start researching pinball AI. This would be similar to current AI research with billiards.

A starting point for this project would be to create platform-independent versions of *Visual PinMAME* and *Visual Pinball*. This would include creating a way to get the ball's location. Some noise model should be applied to the location to prevent agents from having perfect knowledge. The new program could attach the flipper API server described in Section 3.1 and be used to connect AI agents to the simulated world.

If this new program were available, other researchers could attempt a pinball AI without the need for a pinball machine in their office. Hopefully with the simulator running the same game as the physical pinball machine, knowledge of the game could be transfered from the simulator to the real world.

5.1.2 AI Improvements

There are many ways to turn the novice AI agent into a pinball wizard. Section 3.4.2 briefly described some ways to improve the novice AI.

Knowing what to hit is a challenging task to a player that has not seen the game played before. The quests in the game can be complicated and have very different rules between them. Some quests are timed, requiring fine accuracy to complete them. There is no current way to determine what goal the agent should try to complete. A target protocol was described earlier as a way to suggest targets to an agent that should be hit. This allows for higher level planners that choose which shots to attempt.

Without the target protocol, the agent would need to learn what targets to hit based on a score increase or visual change. It is a difficult task to see if a target has been hit or if it was a miss. Combined with the added complication of quests, learning which targets are important is a hard problem.

An agent playing on the Lord of the Rights machine might be able to transfer some of the knowledge it has learned to other machines. The flippers in pinball machines are similar in function. Learned control on one machine might lend itself to control on another.

5.1.3 Framework Improvements

Using an actual pinball machine brings about complications and issues. Extending the current architecture with new features and improvements could make for a stronger AI. Giving higher level languages the ability to interact with the framework could make for stronger agents that can be created with less effort. There are also a few things that the vision system requires that prevent it from working on general pinball games.

Alternate Language Bindings

Programming in C or C++ offers programmers the ability to generate machine code that is heavily optimized. However, it can take longer to create these programs.

The benefits of using scripting/higher-level languages are well known [22]. It is much faster to develop and evolve an application in a scripting language. Higher-level languages such as Ruby, Python, or Java provide built-in constructs and features that can accelerate development time at the cost of execution overhead. These high-level languages provide garbage collection that frees memory from variables when they are no longer referenced. Programmers would not have to worry about memory management and can focus on creating the algorithms and logic of their agents.

The agent's logic is a small part of the framework's execution time. A large majority of the time is spent processing the camera frames. The overhead of using another language for the agent would be small.

To make it easy for language-independent AI agents, the PlayerAPI could be extracted to a socket similar to how the current system interacts with the flipper API. When the ball is located, instead of calling the update function, the system would write a packet to the socket with the current location and velocity. The socket would then read in the commands written by the player. A downside to the socket server model is that it is now much slower than a simple function call.

The other way to allow alternative languages for the pinball AI framework is to generate a module or class in the language that interacts with the C compiled code. With Python, the player could import a PinballAPI module and define an update method to be called at runtime from within the C code. It is possible to create a Python environment, load a file, and execute its functions from within the C code. It is almost as easy as to call Python functions from C as it is to call other C functions. Alternatively, the framework could be compiled into a single shared library that other languages can access.

The second solution would maintain most of the reaction time that is demonstrated with the current framework. The PinballAPI module would be the preferred solution for speed but it requires that that module be ported to each language used. SWIG (Simplified Wrapper and Interface Generator) [12] assists in the generation of modules for different languages. The program takes a generic interface file, which is similar to a C header file, and creates modules native to many different languages automatically. The typical use case for SWIG is to connect a high-level language to a highly optimized shared library whose functions are written in C or C++. The goal of SWIG is to make it easy to connect to libraries to other languages and improve performance of specific functions that might suffer from overhead of the higher-level languages.

The socket solution is easy to implement in different languages. Programmers can interact very easily through the protocol. The agent would be separated from the vision system at the process level which might limit future framework improvements, such as providing the camera image or foreground mask to the agent with updates.

Vision System Improvements

The vision system enforces some rules on the custom pinball games in order to be able to track the ball. The biggest restriction is that lamps must be either on or off, and not change. A smaller limitation of the vision system, but still important, is that it can only track one ball at a time.

Tracking the ball is currently done by detecting motion. The motion is observed as differences in the pixels of the image. The lamps changing will be seen as a difference and need to be filtered before a ball location can be found. In the *Lord of the Rings* pinball machine, the positions of the lamps are constant. The lamps do not move around and therefore could be filtered based on location and their state learned from the pixel intensity or color of that location. Tracking the ball effectively is a challenge and having games without toggling lamps made the problem doable.

The online simulator filtering does a small amount to alleviate the lamps problem. The lamp is a stationary object, opposing gravity, that does not seems like an adequate object to represent the ball, especially when there is a a moving ball in the scene. However, the performance of the agent was hurt with the lamps toggling.

The ball tracking in the vision system uses a simulator to filter a region that a ball can be in. This system works for only a single ball. However, it can be extended to support multiple balls. This would be accomplished by having multiple objects in the simulator creating different regions of interest. The filtering could then act on the union of these regions.

The vision system could be made generic to work on other machines or even older arcade

games, such as PONG or Pac-Man. A robot, created to send input to these games, could make use of the object detection algorithms used in this framework to learn and play the games.

A next step for the vision system would be to remove the limitations placed on the games by returning lamps to the game and allowing for multiple balls. Another direction available for the vision system is to explore other domains that can use the vision system.

Network Server Improvements

As described in Section 3.5, to ensure the quality and fairness of a multi-player game, the architecture will need a change. Industry has been working with latency and time issues for a long time.

Current video game multi-player implementations work in practice but the problem is not completely solved. To increase the smoothness of a game, a client is required to interpolate and continue functioning even if communication with the server has been temporarily lost.

The current multi-player server developed for pinball does not have the robustness to latency that industry has. Sharing a clock is important to determining order of distributed events. Having the server generate ticks at a fixed interval and regularly broadcasting this would generate a clock signal that can be used to synchronize clients. Proceeding with the network sever would involve constructing a better multi-player protocol that connects physical games together, such that the games are capable of playing fairly with each other regardless of network issues.

Expert AI Training

To learn the tools of pinball experts, they could play a special game and be viewed by the framework. Agents could then learn based on what the experts are doing in various states. This might make it easier to learn the skills that the wizards possess. The system could also be used the other way around. By helping novice players pick up these skills sooner, by seeing an AI perform them and attempting to do it themselves.

5.2 Conclusions

There is now a pinball machine that can play itself. Someday in the future, an AI might challenge the high score of a human pinball wizard on his own machine. There are many ways to continue this work and achieve that goal.

Bibliography

[1] Michael Aeberhard, Shane Connelly, Evan Tarr, and Nardis Walker. Single Player Foosball Table with an Autonomous Opponent, 2007. Project Report, Georgia Institute of Technology.

[2] Christopher Archibald, Alon Altman, Michael Greenspan, and Yoav Shoham. Computational Pool: A new challenge for game theory pragmatics. *AI Magazine*, 31(4):33–41, 2010.

[3] Human Benchmark. Human Benchmark - Reaction Time Stats. `http://www.humanbenchmark.com/tests/reactiontime/stats.php` (Accessed May 27, 2011).

[4] Paul Bettner and Mark Terrano. 1500 archers on a 28.8: Network programming in Age of Empires and beyond. *Game Developers Conference*, 2001.

[5] Rodney P. Carlisle. *Encyclopedia of Play in Today's Society*. Sage Publications, 2009.

[6] Erin Catto. Box2D - Home. `http://www.box2d.org`.

[7] Nokia Corporation. Qt - Cross-platform application and UI framework. `http://qt.nokia.com/`.

[8] Valve Corporation. Source Multiplayer Networking - Valve Developer Community. `http://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking`.

[9] Erwin Coumans. Game Physics Simulation. `http://www.bulletphysics.org`.

[10] Mike Cyrus and Jay Beck. Generalized two- and three-dimensional clipping. *Computers & Graphics*, 3(1):23 – 28, 1978.

[11] Randy Davis. The Virtual Pinbal Forums. `http://www.vpforums.org/`.

[12] The SWIG Developers. Simplified Wrapper and Interface Generator. `http://www.swig.org/`.

[13] Steve Ellenoff, Tom Haukap, Martin Adrian, and Gerrit Volkenborn. Welcome to PINMAME. `http://pinmame.retrogames.com/`.

[14] Google. Protocol Buffers - Google's data interchange format. `http://code.google.com/p/protobuf/`.

[15] Michael Patrick Johnson. Algorithms for pinball simulation, ball tracking and learning flipper control. Bachelor thesis, Massachusetts Institute of Technology, 1993.

[16] Pakorn Kaewtrakulpong and Richard Bowden. An improved adaptive background mixture model for realtime tracking with shadow detection. In *Proceedings of the Second European Workshop on Advanced Video-Based Surveillance Systems*, 2001.

[17] Kyungnam Kim, Thanarat H. Chalidabhongse, David Harwood, and Larry Davis. Background modeling and subtraction by codebook construction. In *Image Processing, 2004. ICIP '04. 2004 International Conference on*, pages 3061 – 3064, Oct. 2004.

[18] Will Leckie and Michael Greenspan. Pool physics simulation by event prediction 1: Motion states and events. *ICGA Journal*, pages 214 – 222, 2005.

[19] Fei Long, Johan Herland, Marie-Christine Tessier, Darryl Naulls, Andrew Roth, Gerhard Roth, and Michael Greenspan. Robotic pool: An experiment in automatic potting. In *Intelligent Robots and Systems, 2004. (IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 3, pages 2520 – 2525, September - October 2004.

[20] James MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. California, USA, 1967.

[21] David L. Mills. Internet time synchronization: The network time protocol. *Communications, IEEE Transactions on*, 39(10):1482 –1493, Oct 1991.

[22] John K. Ousterhout. Scripting: Higher level programming for the 21st century. *Computer*, 31(3):23 –30, Mar 1998.

[23] SDL. Simple DirectMedia Layer. `http://www.libsdl.org/`.

[24] Michael Smith. Pickpocket: A computer billiards shark. *Artificial Intelligence*, 171(16-17):1069 – 1091, 2007.

[25] Kenji Suzuki, Isao Horiba, and Noboru Sugie. Linear-time connected-component labeling based on sequential local operations. *Computer Vision and Image Understanding*, 89(1):1 – 23, 2003.

[26] Christopher J.C.H. Watkins. *Learning from Delayed Rewards*. PhD thesis, Cambridge University, 1989.

[27] Willow Garage. Welcome - OpenCV Wiki. `http://opencv.willowgarage.com/wiki/`.

[28] Nathaniel S. Winstead and Alan D. Christiansen. Pinball: Planning and learning in a dynamic real-time environment. In *AAAI-94 Fall Symposium on Control of the Physical World by Intelligent Agents*, 1994.

[29] Daniel Wong, Darren Earl, Fred Zyda, Ryan Zink, Sven Koenig, Allen Pan, Selby Shlosberg, Jaspreet Singh, and Nathan Sturtevant. Implementing games on pinball machines. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games*, pages 240–247, 2010.

[30] Stefan Zickler, Tim Laue, Oliver Birbach, Mahisorn Wongphati, and Manuela Veloso. SSL-Vision: The Shared Vision System for the RoboCup Small Size League. *RoboCup 2009: Robot Soccer World Cup XIII*, pages 425–436, 2009.